Universitat Autònoma de Barcelona
Escola Tècnica Superior d'Enginyeria
Enginyeria Informàtica

GenRegion: aplicatiu de generació de màscares per a la codificació d'imatges amb regions d'interès

Memòria del projecte de final de carrera corresponent als estudis d'Enginyeria Superior en Informàtica presentat per Joan Vidal Plujà i dirigit per Francesc Aulí Llinàs.

Bellaterra, Setembre del 2007

El sotasignat, Francesc Aulí i Llinàs, professor de l'Escola Tècnica Superior d'Enginyeria de la Universitat Autònoma de Barcelona

CERTIFICA:

Que la present memòria ha estat realitzada sota la seva direcció per Joan Vidal Plujà

Bellaterra, Setembre del 2007

Signat: Francesc Auli Llinas

A l'Esther i a la meva família,
per la paciència que han tingut tots aquests anys.

Agraïments

En primer lloc, agrair al Francesc Aulí i al Joan Bartrina l'ajut rebut durant la realització del projecte.

Agraïr també als companys d'universitat, concretament a l'Emili, al David, al Ramon, a l'Àlex i al Cristian. Als companys de feina, especialment al Marcos pel seu ajut i els seus consells. I als amics per haver estat en els bons i els mals moments.

Índex

Capítol 1

Introducció

Aquest projecte, "Generador de Regions", té com a objectiu desenvolupar una aplicació que ens permeti carregar una imatge en format PGM o PPM consumint el mínim de memòria possible i dibuixar-hi regions d'interès, disposant en tot moment d'informació de les regions generades. Posteriorment hem de poder generar una imatge que sigui una màscara de la imatge inicial, visualitzant les regions generades sobre fons negre.

1.1 Antecedents i motivació del projecte

Aquest projecte sorgeix de les necessitats en la creació de màscares i de regions "no-data" que es donen al GICI (*Group on Interactive Coding of Images*), vinculat al DEIC (Departament d'Enginyeria de la Informació i de les Comunicacions) de la UAB (Universitat Autònoma de Barcelona). El grup treballa en diferents aspectes de la compressió i transmissió d'imatges. L'estudi, disseny i implementació dels estàndards i tècniques de codificació i transmissió d'imatges és l'objectiu principal del grup.

GICI ha desenvolupat implementacions Java en codi obert de l'estàndard JPEG2000, el CCDS-IDC, recomanacions per Image Data Coding i tècniques EZW, IC, SPECK, i SPIHT. També treballa en el desenvolupament de noves tècniques per a la transmissió d'imatges basades en el protocol JPIP (JPEG2000 Part 9). Alguns d'aquests esquemes han estat modificats amb l'objectiu d'estudiar noves evolucions o adaptacions d'aquests estàndards.

D'entre les necessitats que han sorgit al GICI a l'hora de treballar amb imatges, ens trobem amb la d'haver de treballar amb imatges de gran tamany i de definir-hi regions d'interès i regions "no data" [1]. L'aplicació ha de permetre trobar fàcilment les regions definides i disposar d'informació de les diferents regions creades, permetent la modificació de les mateixes. D'altra banda, treballar amb imatges molt grans suposa una gran despesa en memòria i un dels requeriments bàsics serà disminuir aquest cost.
Tanmateix, les diferents aplicacions generades des del GICI s'han implementat amb Java donat que totes les aplicacions han de poder ser multiplataforma.

1.2 Problemàtica

El primer problema a solucionar és el d'haver d'obrir imatges, el tamany de les quals pot superar 1Gb de memòria, i fer-ho d'una forma òptima que permeti treballar amb elles d'una forma ràpida i àgil. S'han de poder definir regions de la imatge, permetent dibuixar-les amb el mouse i controlant que les regions creades siguin regions tancades, és a dir les fronteres de les diferents regions creades han d'estar perfectament definides. No s'han de poder generar regions obertes.
Un cop definides les regions hem de poder generar una màscara de la imatge, una imatge del mateix tamany que la imatge inicial però amb fons negre i sobre la qual només s'hi han de veure les regions que haguem definit amb el valor que haguem donat a cadascuna d'elles.

1.3 Solució presentada

Com a solució hem presentat una aplicació implementada amb Java que permet obrir imatges PGM/PPM independentment del tamany d'aquestes. Per tal d'optimitzar la càrrega de la imatge i d'evitar penjar la màquina virtual de Java, optem per carregar només aquella part de la imatge que podem visualitzar per pantalla. D'aquesta forma la càrrega de la

[1] Imatges en les que només hi ha interès per una determinada regió de la imatge.

imatge es fa molt ràpidament donat que haurem de llegir una part relativament petita de la imatge. Això ens permetrà carregar qualsevol tamany d'imatge sense haver de disposar d'un potent maquinari.

Pel que fa a la generació de regions, permetem realitzar diferents tipus de figures i emmagatzemar-les posteriorment en una llista de regions ocupant un mínim d'espai en memòria. Hem disposat una interfície que mitjançant un *JSplitPanel* permet visualitzar ràpidament i còmode les regions creades i d'aquestes passar al dibuix de treball. Hem implementat també un visor on es carrega tota la imatge a una escala reduïda per tenir així un mapa de tota la imatge i accedir ràpidament a qualsevol punt mitjançant un marc de visualització. Donada la variabilitat de tamany de les imatges a obrir, es pot configurar l'escala del visor per què s'adeqüi al tamany de la imatge carregada.

Capítol 2

Anàlisi de requeriments

Fins ara s'utilitzava una aplicació del tipus *gimp* o similar per crear les regions d'interès. Tenia l'inconvenient que no disposaven d'informació sobre les regions definides i el procés de creació de la màscara s'havia de realitzar "manualment" amb les eines que disposés l'aplicació en qüestió.

A continuació farem un repàs dels requeriments funcionals i no funcionals per tal de disposar d'una aplicació que permeti generar fàcilment regions d'interès sobre una imatge, permetent la consulta de les regions existents i facilitant la tasca de creació de la màscara.

2.1 Funcionals

Dins dels requisits funcionals ens trobem amb:

- Disposar d'una interfície que permeti carregar una imatge PGM i/o PPM.
- Poder dibuixar figures sobre la imatge carregada, podent saber en tot moment quantes regions hem dibuixat i on es troben ubicades.
- Totes les figures dibuixades (i conseqüentment les regions posteriors) han de ser figures tancades.
- A mida que anem dibuixant una figura sobre la imatge, aquesta s'ha de desplaçar automàticament quan ens apropem als marges si encara hi ha zones de la imatge per

mostrar.

- Poder dibuixar figures compostes, figures que dins contenen altres figures.
- Poder generar una màscara de les regions dibuixades sobre la imatge d'una forma ràpida i àgil.
- La interfície ha de de permetre treballar amb el màxim de tamany de la imatge possible.
- La interfície ha de disposar d'eines que facilitin treballar amb la imatge, com són zoom i *panning*.
- Disposar d'un visor que permeti tenir un mapa de tota la imatge.

2.2 No funcionals

Dins dels requisits no funcionals, ens hem trobat amb requisits que de bon inici ja estaven contemplats i d'altres que han anat apareixent durant la implementació del projecte. Entre els requisits més importants podem citar:

- Des del GICI demanen la implementació de l'aplicació en Java.
- A l'haver de treballar amb Java i davant la possibilitat de poder obrir imatges molt grans, hem d'optimitzar l'ús de memòria impedint que la màquina virtual Java es pugui penjar i evitar haver de disposar d'un potent maquinari.
- L'aplicació ha de ser multiplataforma.
- Tot i que la prioritat és l'optimització en l'ús de memòria, això no ha de penalitzar el temps de càrrega de la imatge ni les posteriors operacions que es facin amb ella.

2.3 Objectius i problemes a resoldre

Una vegada analitzats els requisits funcionals i no funcionals, en podem extreure quins són els objectius i quins problemes haurem de resoldre. Com a objectius podem definir:

- Disposar d'una interfície que ens permeti carregar imatges PGM i PPM minimitzant el consum de memòria.

- Poder dibuixar figures sobre la imatge i emmagatzemar-les d'una forma òptima. Aquestes figures o regions han de poder ser modificables i s'han de poder esborrar si és necessari.

- Totes les figures dibuixades han de ser figures tancades. Ens hem d'assegurar que sempre quedin tancades independentment de la figura dibuixada per l'usuari.

- Hem de controlar quan ens apropem als límits del contenidor a l'hora de dibuixar, desplaçant la imatge si encara hi ha zones de la imatge per mostrar en la direcció que dibuixàvem.

- Dissenyar una interfície que permeti treballar amb la màxima superfície d'imatge possible i disposant en tot moment de la informació sobre les regions generades.

- Dotar a les interfícies d'utilitats per al treball amb imatges: zoom, *panning*, visor.

Capítol 3

Fonaments teòrics

La base teòrica d'aquest projecte es fonamenta bàsicament en els formats d'imatges amb els que hem de treballar i en les característiques que aporta Java a la programació d'entorns visuals.

3.1 Format d'imatges: PGM i PPM

Tots dos formats són el que es coneix com a formats RAW [1] amb capçalera. Consisteixen en una capçalera amb informació sobre el tipus de imatge, nombre de columnes i nombre de files i a continuació una matriu files*columnes on a cada posició apareix el color per a aquell píxel.

El format PGM (*Portable Gray Map*) [PGM] és dissenyat per tal que sigui extremadament fàcil d'aprendre així com d'implementar aplicacions que en facin ús. Una imatge PGM representa una imatge gràfica en escala de grisos. Per a la majoria d'objectius, una imatge PGM pot ser pensada com un *array* de valors sencers arbitraris.

Cada imatge PGM consisteix en el següent:

1. Un "nombre màgic" per identificar el tipus de fitxer. El "nombre màgic" d'una imatge PGM són els dos caràcters "P5".

[1] En anglés *cru* és un format digital d'imatges en què la imatge es guardada tal i com és captada

2. Espais en blanc (poden ser espais en blanc, tabulacions, CRs, LFs).

3. L'amplada de la imatge, amb format de caràcters ASCII en decimal.

4. Espais en blanc.

5. El màxim valor gris (Maxval), també com a ASCII decimal. Pot ser menor que 65536 i més gran que zero.

6. Una nova línia.

7. Una concatenació de "Alçada" files en ordre de menor a major. Cada fila consisteix en "Amplada" valors de gris en ordre d'esquerra a dreta. Cada valor gris és un número que va de 0 a Maxval on 0 representa el negre i Maxval representa el blanc. Cada valor gris està representat en binari pur per 1 o 2 *bytes*. Si Maxval és menor de 256 és un byte. En cas contrari són dos bytes. El byte més significatiu és el primer.

```
P2
# feep.pgm
24 7
15
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  3  3  3  3  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0 15  0
0  3  3  3  0  0  0  7  7  7  0  0  0 11 11 11  0  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0  0  0
0  3  0  0  0  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

Figura 3.1: Exemple de imatge PGM en format pla.

El format PPM (*Portable Pixel Map*) [PPM], conegut també com a *Portable pixmaps* és un format que permet la representació en color d'una imatge. És un format ineficient en el sentit que representa colors que l'ull humà no pot identificar. Per contra, és un format molt fàcil de programar.

Cada imatge PPM consisteix en el següent:

1. Un "nombre màgic" per identificar cada tipus de fitxer. Una imatge PPM té el "nombre màgic" consistent en dos caràcters "P6".

2. Espais en blanc, que poden ser: espais en blanc, TABs, CRs, LFs.

3. Amplada en format de caràcters ASCII en decimal.

4. Espais en blanc.

5. Alçada, també en ASCII decimal.

6. Espais en blanc.

7. El màxim valor del color (Maxval), també en ASCII decimal.

8. Una nova línia.

9. Una concatenació de "Alçada" files en ordre de menor a major. Cada fila consisteix en "Amplada" píxels, en ordre d'esquerra a dreta. Cada píxel és una terna formada pels valors vermell, verd i blau (RGB).

3.2 Interfícies gràfiques d'usuari amb Java

Els elements bàsics necessaris per crear una GUI (de l'anglès *Graphic User Interface*) [WAL04, ROD01] resideixen en dos paquets *java.awt* i *javax.swing*. El paquet *java.awt* és el primer repositori de classes que s'hauria d'utilitzar per a crear una GUI en Java 1.1 - 'awt' és l'abreviació de *Abstract Windowing Toolkit*, però alguna de les classes que defineix han estat millorades en Java 2 per *javax.swing*. Moltes de les classes del paquet *javax.swing* defineixen elements de GUI, ens referirem a ells com a **components Swing**, els quals proporcionen una alternativa molt millorada als components definits a *java.awt*. De totes maneres, les classes de components Swing generalment deriven, i depenen, fonamentalment de classes del paquet *java.awt*.

Les classes swing són una part d'un conjunt més general de recursos de programació de GUI als que es refereix col·lectivament com a JFC *Java Foundation Classes*. JFC cobreix no només els components Swing sinó també classes per al dibuix en 2D.

Les classes de components Swing són més flexibles que les classes de components definides al paquet *java.awt* donat que es troben implementades totalment en Java. Els components *java.awt* depenen del codi nadiu i estan restringides a un conjunt més baix de capacitats

d'interfície. Donat que els components Swing són pur Java, no estan restringits per les característiques de la plataforma en la que s'estan executant. A part de la funcionalitat i flexibilitat afegides dels components Swing, aquests també proporcionen una característica anomenada *pluggable look-and-feel* que fa possible canviar l'aparença del component.

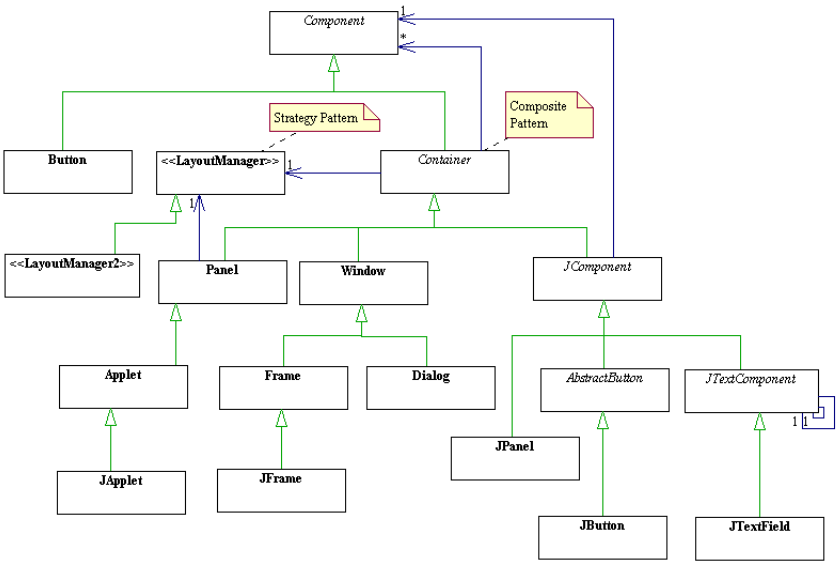


Figura 3.2: Diagrama parcial de components Swing.

Tots els components Swing tenen la classe *JComponent* com a classe base, la qual estén la classe *Component* afegint-li les següents capacitats:

- Suporta *pluggable look-and-feel* per components, permetent canviar l'aparença fàcilment o implementar la pròpia aparença que li volem donar.
- Suporta *tooltips*, és un missatge descriptiu de la funcionalitat d'un component que es mostra quan el cursor del mouse es situa sobre d'ell.
- Suporta el scroll automàtic d'una llista, taula o arbre quan el component és arrossegat amb el mouse.
- Proporciona suport per al testatge, permetent veure què està passant.
- Les classes poden ser fàcilment esteses per crear els nostres propis components.

Totes les classes de components Swing estan definides al paquet *javax.swing* i els noms de les classes comencen sempre amb J. En aquest projecte hem usat components Swing per a la construcció de la interfície.

3.3 El procés de gestió d'events

Les accions realitzades quan s'està utilitzant una GUI són inicialment identificades pel Sistema Operatiu (SO). Per cada acció, el SO determina quin dels programes actualment en execució és el que ha de gestionar l'acció i passa el control a aquest programa. Quan l'usuari prem el botó del mouse, és el SO el que ho registra i anota la posició del cursor del mouse a la pantalla. Llavors decideix quina aplicació controla la finestra on es trobava el cursor quan s'ha premut el botó i comunica aquesta acció al programa. La senyal que el programa rep del SO com a resultat de l'acció és el que anomenem event [HOR00].

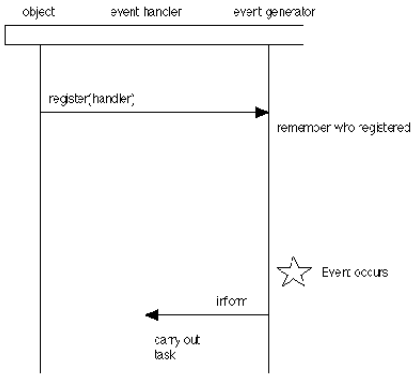


Figura 3.3: Funcionament d'un event.

Per dur a terme la interacció de l'usuari amb els components, hem d'entendre abans com gestiona java els events. Quan l'usuari prem un botó del GUI, aquest botó és la font (o origen) de l'event. L'event generat com a resultat de prémer el botó del mouse és associat amb l'objecte *JButton*. Quan el botó és pitjat, crea un nou objecte que representa i identifica aquest event (en aquest cas un objecte del tipus *ActionEvent*). Aquest objecte contindrà informació sobre l'event i la seva font, i serà passat com a argument al mètode encarregat de gestionar l'event. L'objecte event corresponent al clic del botó serà passat a qualsevol objecte *listener* prèviament registrat com a interessat en aquest tipus d'event. Un *listener* és, doncs, un objecte que escolta només un determinat tipus d'events.

A l'aplicació hem hagut d'implementar diferents *listeners* per a la gestió dels events provinents de la interfície gràfica.

3.4 Càrrega d'imatges amb Swing

En qualsevol aplicació que tracti amb imatges, haurem de mostrar la imatge, manipular-la i en alguns casos imprimir-la. Abans de que la imatge pugui ser interpretada, l'objecte que representi la imatge ha de ser creat. Java proporciona diferents formes de carregar una imatge. En les primeres versions de Java (JDK 1.0 i JDK 1.1) la classe *java.awt.Image* encapsulava una imatge. Però aquesta classe no disposa de mètodes que permetin escriure i llegir píxels directament. Per fer-ho necessitem la classe *PixelGrabber* que ens permet obtenir els valors dels píxels. Per guardar aquestes dades en *cache* s'ha desenvolupat la classe *ImageBuffer*. En *Java2D*, *ImageBuffer* realitza aquesta funció.

Creant un objecte *Image*

Es pot crear un objecte *Image* [JAP] des d'un fitxer local, des d'una URL que apunti a una imatge, des d'un array de bytes o bé aplicant un filtre a una imatge existent. Una imatge és mostrada només quan és dibuixada sobre un context gràfic d'un component visual. La classe *java.awt.Graphics* representa un context gràfic amb diferents mètodes per dibuixar una imatge. Nosaltres utilitzarem:

```
public boolean drawImage(Image theImage, int x, int y, Color bg,
                         ImageObserver obs);
```

Aquest mètode dibuixa la imatge a partir de la posició (x,y). Si la imatge no omple el contenidor, només es mostrarà la part visible.

Components per dibuixar una imatge

Tot i que podem dibuixar imatges sobre qualsevol component visual, només uns quants components són apropiats per aplicar una visualització. Si treballem amb *awt*, podem dibuixar sobre *Canvas*, *Panel*, *Applet* o *Frame*. D'entre totes elles la més apropiada seria *canvas* ja que és descendent directe de *Component* mentre que la resta són contenidors. Només hauríem de sobreescriure el mètode *paint()*.

Mentre que si treballem amb *Swing*, aquest no té equivalent de la classe *Canvas*, però contenidors com *JPanel*, *JFrame* o *JApplet* poden ser utilitzats per dibuixar imatges.

3.5 Manipulació d'imatges amb Java2D

Dibuixar en *components Swing* és una mica diferent que dibuixar en *components awt*. A *Swing*, el mètode *update()* no és mai invocat amb una crida a *repaint()*. En canvi, *paint()* és cridat per a dibuixar tant per l'aplicació com pel sistema. El mètode *paint()* fa molt més que pintar únicament un component. Crida altres mètode com *paintComponent()*, *paintBorder()*, *paintChildren()*. Una aplicació només hauria de sobreescriure el mètode *paintComponent()*, deixant els altres dos disponibles per al sistema. Per a dibuixar tant la imatge, com les figures com les regions, hem sobreescrit únicament aquest mètode.

Descrivint una figura

No hi ha una descripció genèrica d'una figura. Tot i això, la interfície *java.awt.shape* especifica una figura genèrica, la qual es descrita com un camí amb múltiples segments. Les classes que implementen la interfície inclouen *Line2D*, *Rectangle2D*, *Polygon*, *QuadCurve2D* i *CubicCurve2D* [JPA].

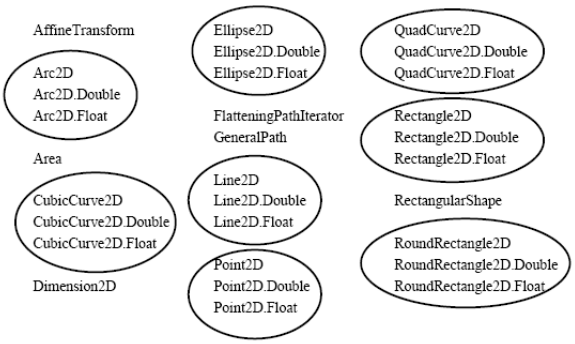


Figura 3.4: Classes de la llibreria java.awt.geom.

La interfície *Shape* inclou els següents mètodes que farem servir:

- **getPathIterator(AffineTransform transform)**: *PathIterator* és una interfície que ajuda a iterar sobre diferents segments de la frontera d'una figura.
- **containes(double x, double y)**: aquest mètode comprova si el punt (x,y) està inclòs dins de la figura.
- **getBounds2D()**: aquest mètode retorna un rectangle que inclou la figura.

En geometria elemental, identificar l'interior d'una regió d'un polígon estàndard, com un triangle, un rectangle o un pentàgon és fàcil perquè no hi ha punts que interseccionin, només els vèrtexs. Quan una figura és complexe, en canvi, determinar els punts de l'interior d'una regió pot resultar difícil. En gràfics hi ha dos regles que són normalment utilitzades per identificar l'interior d'una figura:

- ***Even-odd rule***: aquesta regla especifica que un punt cau a l'interior si una recta dibuixada en qualsevol direcció des d'aquest punt a l'infinit és creuada per segments de la frontera de la figura un nombre senar de vegades.
  Aquesta regla la utilitzarem per a les regions simples.

- ***Nonzero winding rule***: aquesta regla especifica que un punt cau a l'interior de la figura si una recta dibuixada en qualsevol direcció des d'aquest punt a l'infinit és creuada per segments de la frontera de la figura un nombre diferent de vegades en funció de si es fa en el sentit de les agulles del rellotge o si es fa en sentit contrari.
  Aquesta regla la utilitzarem per al dibuix de figures compostes.

Capítol 4

Implementació de la solució

A continuació descriurem les eines que hem triat per a desenvolupar l'aplicació. Enumerarem les raons per les quals les hem triat, argumentant els avantatges que suposen de cara a la implementació de la nostre solució.

Tot seguit detallarem el funcionament intern de l'aplicació, que hem dividit en quatre grans blocs: mòduls de l'aplicació, treball amb la imatge, dibuix de regions i configuració de l'aplicació. Explicarem els motius que ens han portat a prendre cadascuna de les decisions de disseny adoptades.

4.1 Eines utilitzades

L'aplicació s'ha desenvolupat amb la versió de Java 1.5.0. Els motius que ens han portat a optar per Java a l'hora de desenvolupar el projecte són els següents:

- Un requeriment bàsic era que l'aplicació havia de ser multiplataforma. Java permet l'execució en qualsevol tipus de plataforma.
- Per coherència amb la resta d'aplicacions desenvolupades al GICI, que s'han realitzat en Java.
- Per experiència pròpia en aquest llenguatge.
- Perquè existeix una abundant documentació referent a la programació en Java

Com a entorn de desenvolupament hem utilitzat *Eclipse* en la seva versió 3.2. S'ha utilitzat pels següents motius:

- Perquè és un dels entorns de desenvolupament més utilitzats avui en dia, amb tot el que això suposa, abundància de documentació i possibilitat d'utilitzar una gran quantitat de *plugins*.
- Perquè existeixen versions per diferents entorns.
- És gratuït.
- Per experiència personal en aquest entorn.

Com a eina per a la compilació i empaquetat de l'aplicació s'ha utilitzat l'eina *ant*, que ens ofereix les següents avantatges:

- *Ant* [ANH] està implementat en Java i ofereix les mateixes característiques de portabilitat que ens ofereix Java. El nostre fitxer *build.xml* funcionarà correctament en qualsevol entorn on tinguem instal·lat *ant*, ja sigui windows, linux o imac.
- Ens permet definir fàcilment diferents tasques a fer i fixar les dependències existents entre elles.
- Usar ant ens permet no dependre de cap entorn de desenvolupament concret.
- Pràcticament tots els entorns de programació integren o tenen la possibilitat de integrar *ant*, *Eclipse* entre ells.
- A l'igual que *Eclipse* és gratuït.

Per a les proves amb les imatges hem utilitzat el programa *Gimp* perquè existeixen versions per a diferents entorns i és una eina molt estesa avui en dia.

4.2 Mòduls de l'aplicatiu

Inicialment es va pensar l'aplicació com únicament dues pantalles: la pantalla principal on teníem els botons per obrir i guardar la imatge així com la superfície de treball on visualitzàvem les imatges i podíem definir-hi les regions i per altre costat la pantalla del visor global on mostràvem la imatge a una escala reduïda. A l'executar l'aplicació ens apareixia la primera pantalla amb la superfície de la imatge buida, donat que encara havíem de llegir la imatge. Quan obríem la imatge, la mostràvem al panell i obríem una nova pantalla amb el visor global.

Aquesta configuració tenia el problema que quan volíem tancar la imatge per obrir-ne unaltre havíem de tancar l'aplicació i tornar-la a obrir per poder seleccionar una nova imatge. L'efecte d'obrir una gran pantalla sense que hi aparegués res provocava, a més, un efecte d'aplicació poc treballada.

Per corregir aquests aspectes negatius vam pensar en la possibilitat de dividir l'aplicació en tres pantalles diferents:

1. Mòdul principal **GenRegion**: és una petita pantalla que ens apareix a l'executar l'aplicació i que ens permet accedir a les diferents opcions de configuració així com obrir i guardar una imatge.

2. Mòdul de treball **GenRegionFrame**: quan des de la pantalla principal obrim una imatge, es mostra aquesta nova pantalla consistent en un panell on es visualitza la imatge, una botonera amb les diferents opcions de modificació de la imatge i una zona per a l'edició de regions.

3. Mòdul visor **GlobalVisor**: el visor global és una pantalla que ens mostra la imatge oberta a una escala reduïda (en funció de l'escala que haguem configurat). S'obre automàticament a l'obrir una imatge però permet tancar-la i tornar-la a obrir en qualsevol moment.

Aquesta configuració ens permet poder tancar una imatge sense haver de tancar tota l'aplicació i des del punt de vista del disseny és més elegant ja que en cap moment veiem una superfície buida.

A continuació detallarem bàsicament com s'han muntat aquestes tres pantalles a nivell d'interfície, explicant quines operatives gestionen.

4.2.1 Mòdul principal: GenRegion

És la pantalla que ens apareix a l'executar l'aplicació. La seva execució és molt ràpida ja que no realitza cap operació costosa. Disposa d'un menú amb quatre opcions possibles:

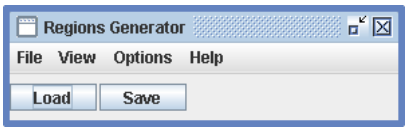


Figura 4.1: Pantalla principal.

1. *File*
   (a) *Load* Ens obre un *FileChooser* que ens permet obrir una imatge.
   (b) *Save* Ens obre un *FileChooser* per seleccionar o escriure el nom del fitxer amb que volem guardar la imatge.
   (c) *Exit* Ens permet sortir de l'aplicació. Si hi ha altres finestres obertes les tanca.
2. *View*
   (a) *PixelMap* Ens obre una pantalla on se'ns permet activar el mapa de píxels seleccionant el color amb el que volem que es mostri.
   (b) *GlobalImage* Ens torna a obrir el visor global, sempre i quan tinguem la pantalla de treball oberta, i per tant, una imatge carregada.
3. *Options*
   (a) *GlobalVisorConfiguration* Ens obre una pantalla on se'ns permet definir l'escala que volem aplicar al visor global.

(b) *SaveFileOptions* Ens permet seleccionar entre dues opcions possibles a l'hora de desar la imatge: la imatge tal qual la veiem per pantalla o bé la màscara de les regions definides.

4. *Help*

(a) *Help Contents* Ens obre el manual d'usuari en format PDF (*Portable Document File*).

(b) *Help about* Ens mostra una petita pantalla amb informació sobre l'aplicació.

El botó *Load* equival a l'opció del menú del mateix nom.
El botó *Save* equival a l'opció del menú del mateix nom.
El punt d'entrada de l'aplicació, i per tant on trobarem el mètode *main* es troba a la classe *GenRegion*. Des del mètode *main* invoquem al constructor, des d'on cridem al mètode *createGUI* on creem un *JFrame* amb uns tamanys predefinits i no modificables i el situem a la cantonada superior esquerra. Carreguem els components del menú *loadMenuOption()* i hi afegim els botons *loadButtons()*.

En el *listener* controlem quin event s'ha llançat, cridant a l'acció pertinent. A la següent taula podem veure els diferents events existents i les accions associades.

Event	**Acció a realitzar**
buttonLoad o *menuLoad*	Carreguem la imatge seleccionada
buttonSave o *menuSave*	Guardem la imatge carregada
Exit	Tanquem l'aplicació
mGlobalVisorConf	Configuració del visor
mSaveFileOpt	Configuració de l'opció d'emmagatzematge
mPixelMap	Configuració del mapa de píxels
mView	Mostrem el visor global
menuHelp	Mostrem el manual d'usuari

Quan obrim una imatge, venim de l'event generat pel botó *Load* o bé per l'opció del menú del mateix nom, si hem pitjat el botó *Accept* de la pantalla de selecció de fitxers, llavors

cridem al mètode: *loadImage()* que el que fa és crear un nou objecte de la classe *GenRegionFrame* des del qual es mostrarà la imatge seleccionada.

Quan hem cridat a l'opció de guardar una imatge, si hem pitjat el botó *Accept* i després de comprovar que la extensió sigui correcte, cridem al mètode *saveImage* passant com a paràmetre el nom del fitxer que li hem donat i generem un fitxer amb el format que correspongui en funció de la configuració existent (guardar màscara o bé guardar imatge).

Si hem activat l'opció del menú *GlobalVisorConfiguration* creem un objecte del tipus *GlobalVisorConf* que llançarà la pantalla de configuració del visor global, on podrem determinar l'escala a aplicar al visor.

Si hem activat l'opció del menú *SaveFileOptions* creem un objecte del tipus *SaveFileConf* que llançarà la pantalla de configuració de les opcions d'emmagatzematge de la imatge, que són dues: guardar la imatge com a màscara o bé tal i com la veiem per pantalla.

A l'activar l'opció del menú *PixelMap* creem un objecte del tipus *PixelMapConf* que llançarà la pantalla de configuració del mapa de píxels, que ens permet dibuixar una graella que ens diferenciï els píxels uns dels altres.

Tot i que la imatge es carrega i es visualitza des de la classe *GenRegionFrame* la lectura de la imatge es realitza des d'un mètode present en aquesta classe. Això és així perquè la lectura de la imatge ha de sincronitzar el desplaçament del marc del visor global amb el desplaçament de la imatge en el marc de treball.

4.2.2 Mòdul de treball: GenRegionFrame

Un cop seleccionada la imatge a llegir des de l'aplicació principal, es llança la creació d'aquesta nova pantalla. És on visualitzarem la imatge i des d'on crearem les diferents regions d'interès.

Inicialment havíem pensat la pantalla com una pantalla redimensionable dividida en dos panells. Un a l'esquerra on apareixeria la imatge carregada i unaltre a la dreta on apareixeria la informació relativa a les regions així com una utilitat de posicionament a la imatge. Aquesta implementació tenia l'avantatge que podíem veure en tot moment la informació de les diferents regions creades, però al mateix moment perdíem espai de visualització de la imatge.

Per superar aquest impediment, vam pensar en afegir un *JSplitPane* [JSP] que ens permetés dividir la pantalla en dues zones: esquerra i dreta. D'aquesta forma l'usuari podrà accedir ràpidament a la visualització de les regions o bé guanyar espai de visualització amb un sol clic.

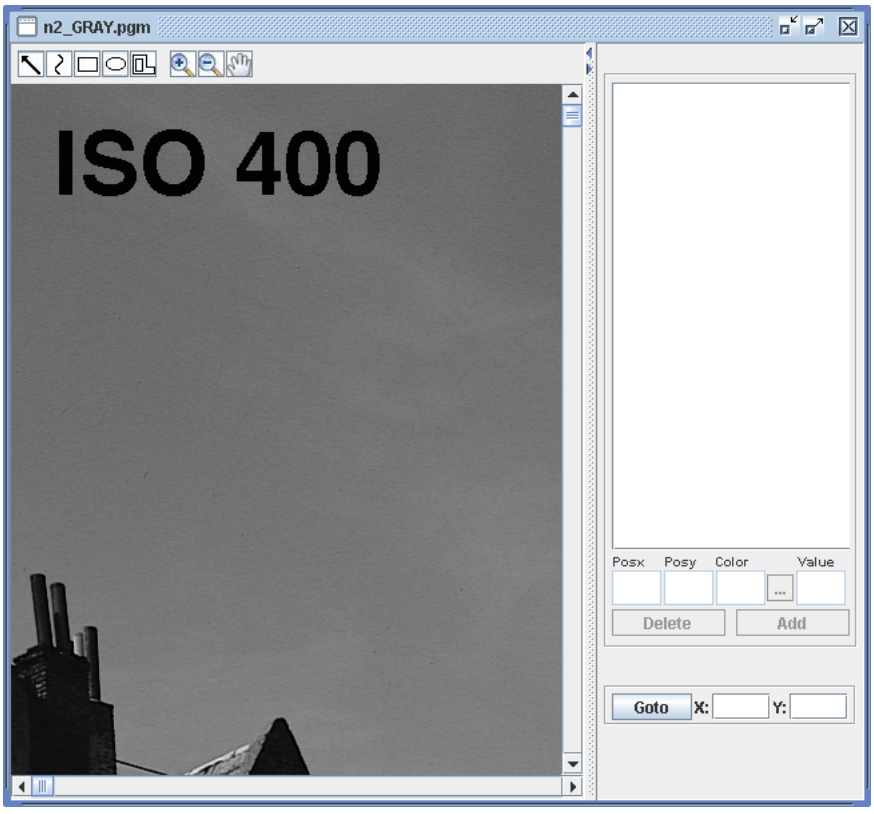


Figura 4.2: Pantalla de treball.

A la figura 4.2 podem veure clarament les dues zones separades pel *JSplitPane*. A l'esquerra la imatge i a la dreta la llista amb les regions existents. Si volem guanyar espai de visualització podem amagar la visualització de les regions i recuperar-la quan vulguem. El mètode constructor de la classe *GenRegionFrame* és on creem la interfície gràfica corresponent a

aquesta pantalla, que ho hem dividit en els següents mètodes:

1. *createGlobalInterface()* on creem la divisió de la pantalla mitjançant un *JSplitPane*, que divideix la finestra en dues parts: esquerra i dreta. A l'esquerra és on tenim la botonera i el panell de visualització de la imatge. A la dreta mostrem els components per a la gestió de les regions.

2. *createRegionsInterface()* on creem i posicionem els components de la dreta del *JSplitPane* que ens serveixen per a la creació, modificació i eliminació de les regions, així com una utilitat per al posicionament en pantalla a partir d'una posició. Les regions les mostrarem en un *Jlist*.

3. *createUtilitiesInterface()* on creem la botonera de la part superior de la imatge on apareixeran els botons per al dibuix de les regions i els de zoom i panning de la imatge.

4. *createDisplayImageInterface()* on creem el *JPanel* on visualitzarem la imatge. Per implementar-ho hem creat una classe *ImageVisor.java* que estén *JPane* i on es centralitza la gestió del dibuix per pantalla, ja sigui de la imatge o d'una regió.

Un cop creada la interfície fem una crida al mètode *readAndDisplay* passant com a paràmetres l'origen (si és del visor global o de la imatge), el nom del fitxer, la posició inicial, l'amplada i l'allargada del que volem llegir de la classe *GenRegion* que és on es centralitza la lectura de la imatge (accés al fitxer per llegir les posicions que volem mostrar per pantalla).

Assignem una acció diferent en funció del botó pitjat. Ho assignem a un atribut privat de tipus sencer, *action*. Aquest valor serà accessible des de tota l'aplicació a través dels seus mètodes públics. Els seus valors disponibles els podem veure a la taula següent:

Acció	**Valor**	**Acció que realitza**
Nothing	0	Mostrem informació dels píxels
Draw	1	Dibuixem a mà alçada
Rectangle	2	Dibuixem un rectangle
Ellipse	3	Dibuixem una El·lípse
Composed shape	4	Dibuixem una figura composta
Zoom-In	10	Apropem la imatge
Zoom-Out	11	Allunyem la imatge
Panning	12	Ens permet arrossegar la imatge amb el mouse

Des d'aquesta classe es gestionen els diferents events del *scroll*, del *panning*, del redimensionat de la pantalla i dels diferents *JButton* i *JTextField* existents. Per a la gestió dels events del *JSplitPane* hem hagut d'implementar la classe *MyJSplitPane* que estén de *JSplitPane*. Per al cas del *scroll*, el *panning*, el redimensionat i el *JSplitPane* el que fem és calcular quines són les noves dimensions i quina posició inicial té i cridar al mètode de lectura de la imatge existent a la classe *GenRegion*.

4.2.3 Mòdul visor: GlobalVisor

El Visor global ens permet tenir una visió general de tota la imatge, amb un marc que ens permet desplaçar-nos ràpidament per la imatge i que està sincronitzat amb la imatge de la finestra de treball. La implementació del visor es troba dividida en diferents classes:

- *GlobalVisor* que estén de *JFrame*, s'encarrega de crear la pantalla.
- *VisorPanel* que estén de *JPanel*, s'encarrega de la creació d'un *JPanel* on es visualitzarà la imatge a escala. És en aquest mètode on llegim la imatge a mostrar.
- *Panner* que estén de *JComponent* és on creem el marc que podrem moure per tota la imatge global i que estarà sincronitzat amb la zona de la imatge visible de la pantalla de treball.

Per poder mostrar una imatge a escala hem de llegir tota la imatge origen. Inicialment recorríem tota la imatge per trobar els punts a mostrar i això comportava un cost signifi-

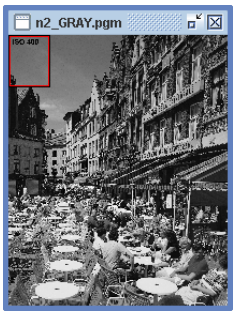


Figura 4.3: Visor Global d'una imatge PGM.

catiu de temps. Hem solucionat aquest problema saltant les posicions de la imatge que no hem de llegir, utilitzant el mètode *skipBytes()* per saltar els bytes que no hem de mostrar, calculats a partir de l'escala de visualització. Hem de tenir en compte, que la lectura de la imatge global només la fem quan carreguem la imatge i sempre que tornem a obrir el visor global.

El mètode de lectura el tenim a la classe *VisorPanel*. Llegim la imatge a escala i la guardem en una matriu de *floats*. El mètode *displayImage* de la mateixa classe, s'encarrega de passar de la matriu a un *BufferedImage*. A la classe *Panner* convertim el *BufferedImage* a un *Image* dins del mètode *paintComponent* d'aquesta mateixa classe. Tant el dibuix de la imatge a escala com el del marc de visualització es realitzen dins d'aquest mètode.

4.3 Treball amb la imatge

A continuació ens centrarem en l'explicació del funcionament de les diferents operacions que podem fer amb la imatge, fent esment dels problemes amb que ens hem trobat i la forma com els hem solucionat. El fet que incloem en aquest punt aspectes com el zoom i el panning és degut a que de fet, no són més que una nova lectura de la imatge amb unes noves posicions inicials i/o uns nous tamanys.

4.3.1 Lectura de la imatge

Al treballar amb Java i haver d'obrir imatges de gran tamany, ens podíem trobar amb què la màquina virtual de Java fos incapaç de gestionar una quantitat tant gran de memòria. Davant d'aquesta possibilitat vam apostar per reduir el tamany d'imatge amb que treballaríem en memòria. En lloc de carregar tota la imatge, el que fem és carregar només aquella part de la imatge que podem visualitzar per pantalla. Inicialment tenim que la configuració del visor és de 650*600 píxels amb el que la superfície apta per a mostrar la imatge és de 412*517 píxels. El que fem és, doncs, llegir únicament els primers 412*517 píxels de la imatge, sigui quin sigui el tamany de la imatge. Només en cas que la imatge a mostrar fos més petita que el contenidor que la mostrarà llegiríem una quantitat menor de píxels.

Amb aquesta mesura aconseguíem reduir força el temps de lectura i la memòria consumida però continuàvem fent (amplada*alçada) accessos al fitxer de la imatge per llegir el valor del píxel per a cadascuna d'aquestes posicions. Per disminuir el nombre d'accessos al fitxer, llegim tota l'amplada per a una y donada, i sense fer cap accés més al fitxer tractem tots els píxels d'aquesta y. Posteriorment ens saltem els píxels corresponents a la fila y que ens queden fora del contenidor. D'aquesta forma passem de fer (amplada*alçada) accessos al fitxer a fer únicament amplada accessos. Inicialment quan el tamany del contenidor és de 412*517 píxels, estem passant de fer 412*517 accessos a fer-ne únicament 517. A continuació podem veure el codi on realitzem la lectura del fitxer quan es tracta d'una imatge PGM. En aquest cas el color corresponent al píxel és el mateix per a l'escala de vermell,

verd i blau del RGB. Quan llegim una imatge en color el valor de cadascun d'aquests colors serà diferent.

```
(...)
byte[] aBytes;
for(int y=0;y<height;y++){
  aBytes = new byte[width];
  int numBytes = dStream.read(aBytes);
  for(int x=0;x<width;x++){
    pixel = (float)aBytes[x];
    for(int c=0;c<imagePO.getChan();c++){
      imageR[c][x][y]= pixel;
    }
  }
dStream.skipBytes(imagePO.getOrImageWidth()-width);
}
(...)
```

Dins la classe *GenRegion* és des d'on es centralitza la lectura de la imatge. Això és així, ja que la crida per una nova lectura de la imatge pot venir de la pantalla de treball o bé del visor global. El punt d'entrada és el mètode *readAndDisplay* al que li passem els següents paràmetres:

- *isVisor*: valor booleà que indica si la crida prové de la finestra de treball o bé del visor global. isVisor = true indica que prové del visor global, en cas contrari prové de la finestra de treball.
- s: cadena de caràcters que representa el nom del fitxer de la imatge.
- x0: sencer que identifica el punt X inicial a mostrar.
- y0: sencer que identifica el punt Y inicial a mostrar.
- *width*: sencer que identifica l'amplada a llegir de la imatge.

- *height*: sencer que identifica l'alçada a llegir de la imatge.
- zoom: sencer que representa el nivell de zoom a aplicar.

Inicialment, el primer cop que accedim a la imatge, creem l'objecte *ImagePO*, una classe pròpia on emmagatzemar els atributs importants de la imatge llegida, com són:

- *path*: cadena de caràcters que representa el nom i ubicació de la imatge.
- *imageWidth*: amplada de la imatge, pot variar en funció del nivell de zoom aplicat.
- *imageHeight*: alçada de la imatge, pot variar en funció del nivell de zoom aplicat.
- *orImageWidth*: amplada inicial de la imatge, és invariable.
- *orImageHeight*: alçada inicial de la imatge, és invariable.
- *imgWidthOnScreen*: amplada de la imatge a la pantalla.
- *imgHeightOnScreen*: alçada de la imatge a la pantalla.
- *off*: sencer que representa el número de bytes des de l'inici del fitxer fins l'inici dels valors dels píxels de la imatge.
- *imageRGB*: matriu de flotants que contindrà la part de la imatge que mostrem per pantalla en cada moment.
- *imgType*: utilitzem per indicar el tipus d'imatge llegit.

La lectura de la imatge pot provenir del visor global (hem desplaçat el marc de visualització) amb la qual cosa hem de rellegir la imatge a partir de les noves coordenades inicials calculades a partir de la posició del marc o bé de la finestra de treball, ja sigui perquè hem redimensionat la pantalla, perquè hem desplaçat el scroll o perquè hem mogut la imatge amb el *panning*. En aquest darrer cas, el que hem de fer és calcular la posició del marc en funció de les noves coordenades de la imatge. La lectura de la imatge està implementada en tres mètodes diferents, en funció de si es realitza sense zoom aplicat, amb zoom positiu o bé amb zoom negatiu.

Un cop carrega la matriu del que podem visualitzar de la imatge, cridem al mètode *displayImage* on generem un *BufferedImage* a partir de la matriu. La visualització es realitza a la classe *ImageVisor* generant una imatge a partir del *BufferedImage* i mostrant-la dins del mètode *paintComponent*.

Sempre que estem visualitzant la imatge aplicant un zoom-out o sense zoom aplicat i pitgem el botó *clean*, podrem comprovar el color i la posició de cadascun dels píxels simplement passant amb el mouse per sobre de qualsevol punt de la imatge. Aquesta utilitat l'hem implementat amb *JToolTips* [JTO], que permet visualitzar un *hint* [1] informatiu al posicionar-nos amb el mouse a sobre d'un component.

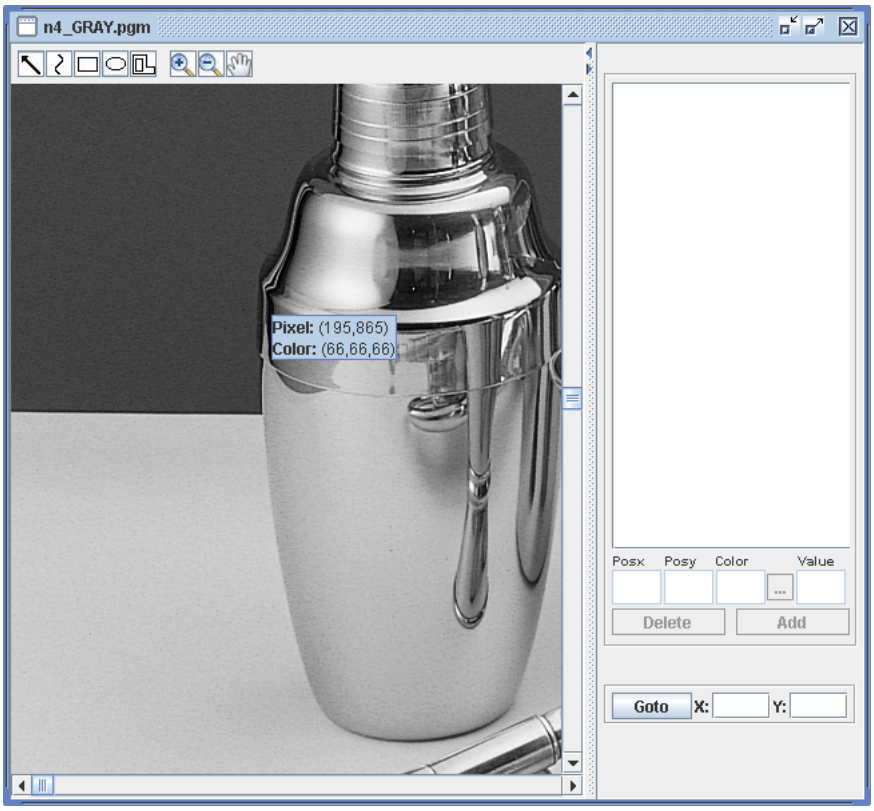


Figura 4.4: Missatge informatiu al posicionar-nos sobre la imatge.

A la figura superior podem veure l'efecte produït al posicionar-nos amb el mouse sobre qualsevol punt de la imatge. Apareix un rectangle informant-nos del color i de la posició del píxel. Per a poder veure aquest hint informatiu hem de tenir seleccionada la pantalla de treball i estar treballant sense zoom o bé amb zoom-out. Amb zoom-in no té sentit

[1] Missatge informatiu que ens apareix quan ens posicionem sobre un component.

disposar d'aquesta utilitat.

4.3.2 Zoom de la imatge

A l'objecte *ImagePO* guardem a l'atribut zoom el nivell de zoom aplicat. Amb zoom=0, com és inicialment, no hi cap nivell de zoom aplicat. Cada vegada que apliquem un *zoom-in*, això és aplicar un zoom positiu o fer la imatge més gran, sumarem 2 a l'atribut zoom. Cada vegada que apliquem un *zoom-out*, això és aplicar un zoom negatiu o fer la imatge més petita, restarem 2 a l'atribut zoom.

A partir d'aquí, quan l'atribut *imagePO.zoom* tingui un valor >0, aproparem la imatge, quan tingui un valor negatiu ens allunyarem de la imatge i en cas contrari farem una lectura de la imatge sense ampliacions ni reduccions.

imagePO.zoom	Acció a realitzar
$= 0$	Llegim imatge sense modificacions
> 0	Llegim imatge aplicant un zoom positiu (ens apropem)
< 0	Llegim imatge aplicant un zoom negatiu (ens allunyem)

Per al cas de zoom positiu, calcularem quina és la nova posició inicial i cridarem a llegir les noves dades de la imatge a mostrar per pantalla. L'algorisme que es segueix consisteix en mostrar *imagePO.zoom* vegades cada píxel. Per tant necessitem llegir del fitxer *imagePO.zoom* píxels menys dels que hauríem de llegir per a una visualització sense zoom. Cadascun dels píxels llegits el repetirem *imagePO.zoom* vegades. Per al cas del zoom negatiu, haurem de llegir fins a *imagePO.zoom* de píxels més per tal de mostrar una representació més àmplia de la imatge.

Cada vegada que apliquem un determinat nivell de zoom, també redimensionem el tamany del marc de visualització del visor global per tal d'adequar-lo a les noves dimensions de la imatge. També provoca el redimensionat de les barres de l'scroll, que s'han de reconfigurar en funció del nou tamany de la imatge. Modifiquem el seu valor màxim en funció del "nou tamany" de la imatge al contenidor de la imatge.

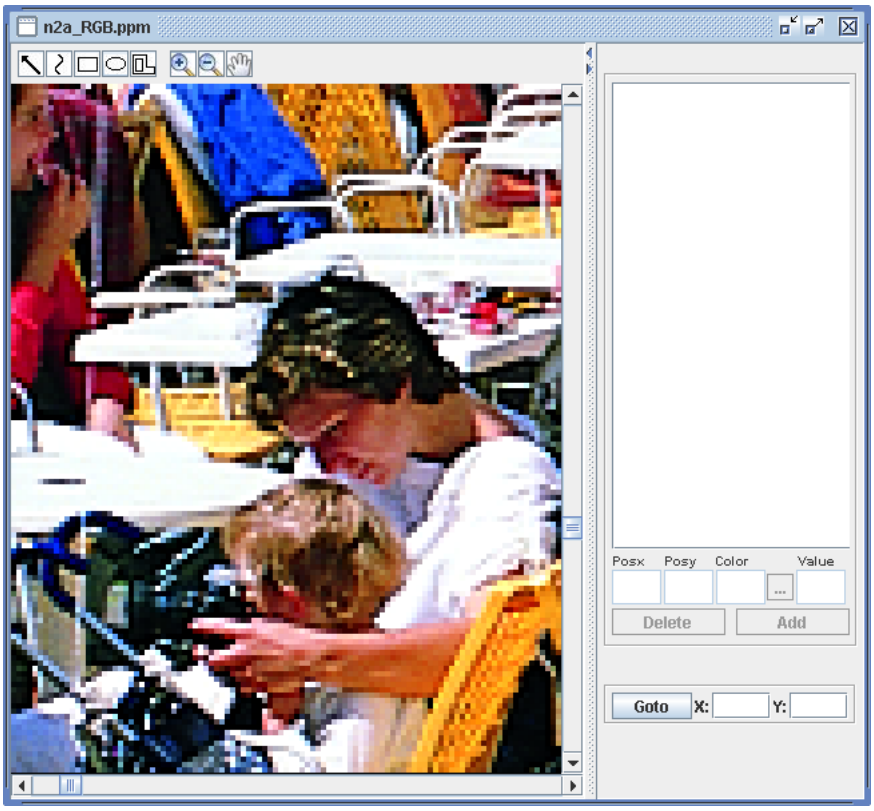


Figura 4.5: Imatge PPM amb zoom-in aplicat.

4.3.3 Panning i scroll de la imatge

Totes dues opcions tenen el mateix resultat, provoquen el desplaçament de la imatge en un o altre sentit. És per aquest motiu que les tractem en el mateix punt.

- *Panning*: inicialment ho vam configurar redibuixant la imatge al capturar l'event *mouseDragged*, és a dir, a l'arrossegar la imatge. La imatge es desplaçava però es produïa un molest efecte de parpelleig. A continuació es va provar únicament capturant els events de *mousePressed* i *mouseReleased*, és a dir el moment en què es premia el botó esquerra i el moment en què es deixava anar. Es calculava la diferència de posicions i a partir d'aquesta diferència es movia la imatge. Aquest procés funcionava correctament però tenia l'inconvenient que no es veia la imatge desplaçant-se. Només es movia el punter del mouse i quan el deixaves anar es redibuixava la imatge a la nova posició. Finalment vam trobar la solució gràcies a la tècnica del *DobleBuffering* [DBU] que permet visualitzar una imatge en desplaçament sense efecte de parpelleig gràcies a realitzar una escriptura prèvia en memòria fora del mètode *paintComponent*.

La gestió del *panning* es realitza des de la classe *ImageVisor*. En l'event *mousse-Pressed*, que es produeix quan l'usuari clica sobre la imatge de treball, guardem en el cas que l'acció en curs sigui la corresponent al *panning* la posició del mouse en el punt on ha clicat.

Disposem d'un atribut (dragging) per controlar quan s'està en el procés d'arrossegament i quan no. Mentre s'estigui arrossegant el mouse, anirem capturant l'event *mouseDragged*, calculant la nova posició i redibuixant la imatge.

Finalment hem implementat un nou mètode al qual es crida des del mètode *paint-Component* per tal d'implementar la tècnica del *doubleBuffering* i poder seguir el desplaçament de la imatge al mateix moment que l'estem movent.

Com en qualsevol moviment de la imatge, desplacem també les barres de scroll i el marc de visualització del visor global. Aquest procés es realitza independentment del nivell de zoom aplicat.

- scroll: el scroll el gestionem des de la classe *GenRegion*. Per al scroll la única cosa que fem és calcular el que hem pitjat sobre la barra tenint en compte que les barres de scroll estan configurades amb els següents valors:

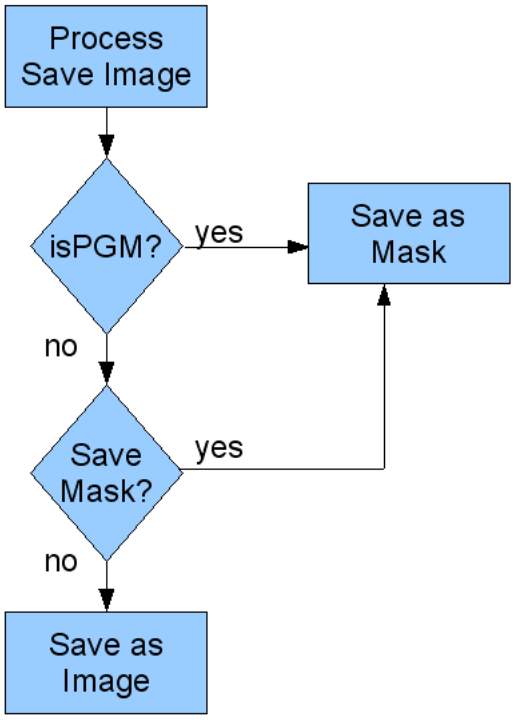
Acció	desplaçament
Pitgem el botó de la *JScrollBar*	1 píxel
Pitgem la barra del *JScrollBar*	10 píxels

Utilitzem un atribut *ScrollListener scrollList = null;* que utilitzem per controlar quan volem capturar els events produïts per l'scroll i quan no, ja que en cas contrari, el desplaçament del marc del visor provocava un desplaçament a la imatge que modificava la ubicació de les barres del *JScrollBar* que tornava a produïr un desplaçament a la imatge. D'aquesta forma, sempre que hem de modificar la posició de les barres de l'scroll, removem el *listener* de les barres i un cop realitzat el desplaçament el tornem a assignar.

4.3.4 Emmagatzematge de la imatge

Com a requeriments de l'aplicació havíem de generar una màscara de les regions dibuixades, on a l'interior de la regió havia d'aparèixer el valor assignat a la regió en el moment de crear-la. Però en el moment de la implementació vam trobar interessant poder guardar la imatge tal qual la veiem per pantalla per poder realitzar comprovacions amb la màscara. Per tant s'ha introduït la possibilitat de triar una opció o unaltre. Tot i això, l'opció per defecte és guardar la màscara de la imatge amb les regions creades.

- *Save Mask With Regions*: amb aquesta opció generem la màscara de les regions definides a la imatge. Generem una imatge de color negre de les mateixes dimensions que la imatge original, però en la posició de cadascuna de les regions hi pintem el seu valor. Amb aquesta opció, generem sempre una imatge PGM. Si indiquem un nom de fitxer amb qualsevol altre extensió ens apareixerà un error. El nom del fitxer de sortida també ha de ser diferent del d'entrada, per evitar sobreescriure el fitxer font i perdre les dades d'origen.



- *Save Image With Regions*: amb aquesta opció generem una imatge idèntica a la que tenim en aquests moments en pantalla. És a dir, la imatge original més totes les regions generades. Aquí, a diferència de la primera opció, guardem les regions amb el

color assignat, enlloc de amb el valor, ja que el que ens interessa és poder disposar de la imatge tal qual l'hem deixat. Com no tenim tota la imatge carregada en memòria, hem de recórrer tot el fitxer d'origen per poder generar la imatge de sortida.

4.4 Dibuix de regions

A nivell d'aplicació fem una diferenciació entre la figura que dibuixem i la regió generada. Ho fem així per permetre triar la figura correcta que volem passar a regió. D'aquesta forma quan pitgem un d'aquests botons (*Draw*, *Rectangle*, *Ellipse* , *Complex Shape*), s'activa la selecció de color i el *textfield* per a l'assignació del valor. A partir d'aquest moment podem seleccionar el color amb el qual volem fer el dibuix. El dibuix que fem serà sempre un dibuix tancat. El tancament de la figura el realitza l'aplicació automàticament quan deixem anar el botó del mouse. En aquest moment tenim una figura tancada, però no és encara una regió. Si en aquest moment tornem a dibuixar sobre la imatge perdrem la figura que teníem anteriorment. Si volem convertir la figura en regió, hem d'assignar-li un valor (1-255) i prémer el botó *Add Region*. En aquest moment es calculen tots els punts de l'interior de la figura i es mostren per pantalla.

A nivell de codi hem implementat una classe diferent per a cada tipus de figura possible. Ho hem fet així per controlar exactament el primer punt del dibuix i per permetre ampliar funcionalitat en un futur en cas que vulguem fer qualsevol mena de modificació. Amb les classes: *SimpleShape*, *SimpleEllipse*, *SimpleRectangle* i *ComposedShape* és com controlarem el dibuix de cadascuna de les diferents figures que podem realitzar.

4.4.1 Línies

Amb la classe **SimpleShape** controlarem el dibuix de figures a "mà alçada" [2]. Són aquests dibuixos de forma indefinida que farem simulant dibuixar a mà alçada amb l'ajut del mouse.

A la classe, hem definit els següents atributs:

- *start*(de tipus *Point2D*): l'utilitzem per indicar quin és el primer punt de la figura creada.
- *paht* (de tipus *GeneralPath*): l'utilitzem per emmagatzemar l'estructura de punts de la figura. L'utilitzarem posteriorment per calcular quins punts queden a dins i quins a fora de la figura dibuixada.

[2]Ens referim als dibuixos que podem fer lliurament sobre la imatge usant el punter del mouse.

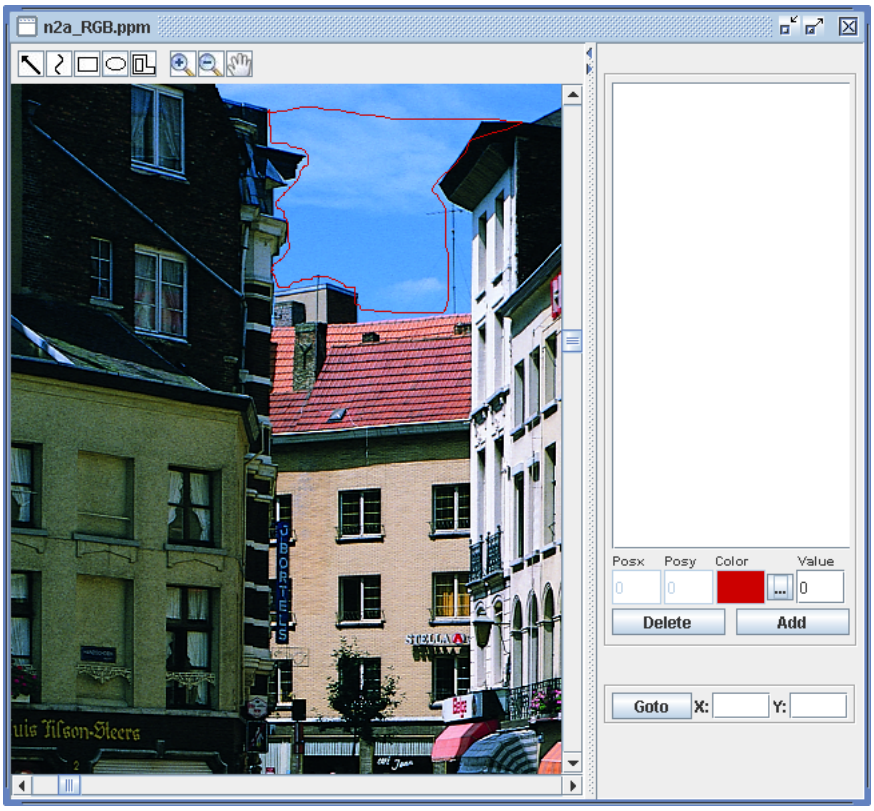


Figura 4.6: Figura generada amb línia.

- *color* (de tipus *Color*): l'utilitzarem per indicar el color associat a la figura. És a dir, el color amb el qual l'hem dibuixada.

- *points* (de tipus *LinkedList*): l'utilitzem per anar guardant els punts de la figura. Ho fem servir per tancar la figura quan estem dibuixant amb un nivell de zoom-in aplicat.

Des de la classe **ImageVisor** controlem quan comencem a dibuixar en funció de l'acció configurada i de quan premem el mouse a sobre de la figura. En aquest moment creem l'objecte *SimpleShape*. El que fem és assignar el punt *start* i cridar al mètode *createShape* on inicialitzem l'atribut *path* creant un *GeneralPath* de tipus *WIND_NON_ZERO* [HOR00]. Es seguirà aquesta regla a l'hora de calcular els punts de l'interior. Per situar el principi del segment, usarem el mètode *moveTo* de *GeneralPath* que ens permet situar l'inici del *GeneralPath* al punt que li passem. Cada vegada que ens desplacem per la pantalla, cridarem al mètode *addPointToShape* que ens farà una línia de l'anterior punt del *GeneralPath* a l'actual.

Quan haguem de tancar la regió, ho controlem des de *ImageVisor* capturant l'event en que deixem anar al mouse, cridant al mètode *closeShape()* que invoca al mètode *closePath* propi de la classe *GeneralPath*, tancant la figura realitzant una línia entre un punt i l'altre. Només s'ha de controlar en el cas d'estar dibuixant amb un nivell de zoom positiu. En aquest cas, hem de calcular els punts manualment cridant al mètode *closePositiveZoomShape*, ja que hem de controlar que no es pugui realitzar un segment que no es correspongui amb els píxels reals de la imatge.

4.4.2 Rectangles

Amb la classe **SimpleRectangle** controlarem el dibuix de figures de tipus rectangle. A mida que ens desplacem amb el mouse, anirem modificant les dimensions del rectangle dibuixat. És una classe pròpia que estén la classe de Java *Rectangle2D*. Ho fem així per permetre la possibilitat d'afegir funcionalitat sense haver de realitzar excessius canvis en el codi.

- *initialPoint* (de tipus *Point2D*): l'utilitzem per emmagatzemar el punt inicial del dibuix.
- *color* (de tipus *Color*): usat per emmagatzemar el color de la figura.

Des de la classe **ImageVisor** controlem quan comencem a dibuixar en funció de l'acció configurada i de quan premem el mouse a sobre de la figura. En aquest moment creem l'objecte *SimpleRectangle*. Cridem al constructor de la classe superior i, a continuació, assignem el punt inicial de la figura.

Des de la classe *ImageVisor*, tant als events *mousePressed*, com al *mouseDragged* com al *mouseReleased* comprovem quina és la nova amplada i alçada per redimensionar el tamany de la figura.

4.4.3 El·lipses

Amb la classe **SimpleEllipse** controlarem el dibuix de figures de tipus el·lipses, ja siguin el·lipses pròpiament o bé cercles. A mida que ens desplacem amb el mouse anirem modificant

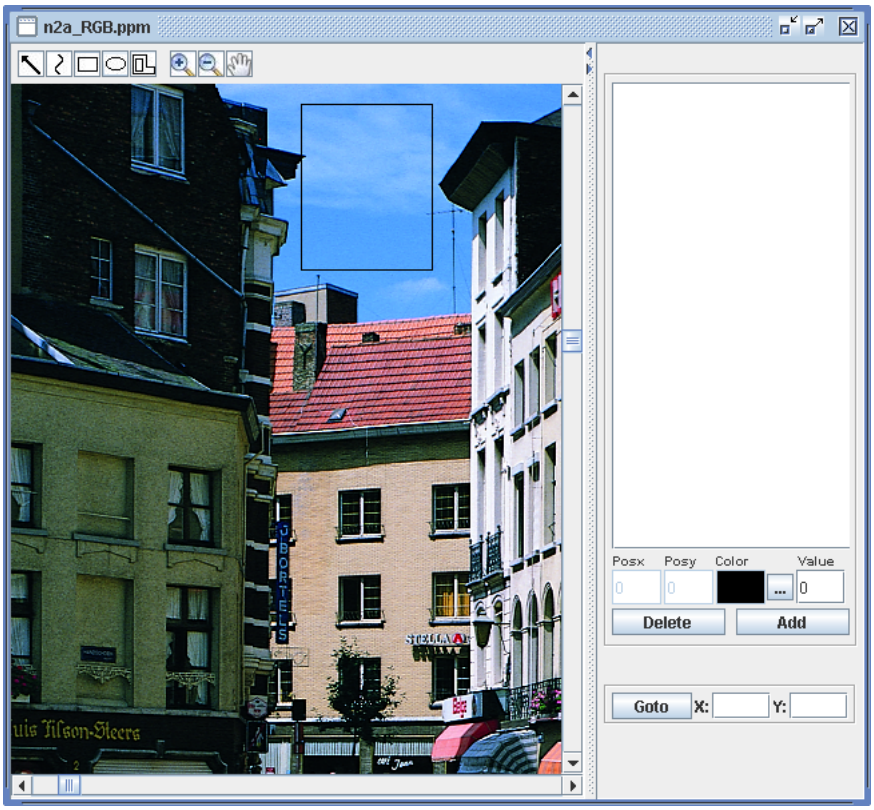


Figura 4.7: Figura generada amb la utilitat Rectangle.

les dimensions de l'el·lipse dibuixada.

- *initialPoint* (del tipus *Point2D*): l'utilitzem per guardar el punt inicial de la figura.
- *color* (del tipus *Color*): l'utilitzem per guardar el color de la figura.
- *ellipse* (del tipus *Ellipse2D*): l'utilitzem per generar l'estructura del dibuix. Usarem el mètode *setFrame* per reassignar l'el·lipse cada vegada que la redimensionem al dibuixar.

Des de la classe **ImageVisor** controlem quan comencem a dibuixar, en funció de l'acció configurada i de quan premem el mouse a sobre de la figura. En aquest moment creem l'objecte *SimpleEllipse*:

En aquesta classe, veiem que enlloc d'estendre de *Ellipse2D*, hem inclòs un atribut d'aquest tipus, per així poder usar el mètode *setFrame* per redimensionar la figura. Des de la classe *ImageVisor*, tant als events *mousePressed*, com al *mouseDragged* com al *mouseReleased* comprovem quina és la nova amplada i alçada per redimensionar el tamany de la figura.

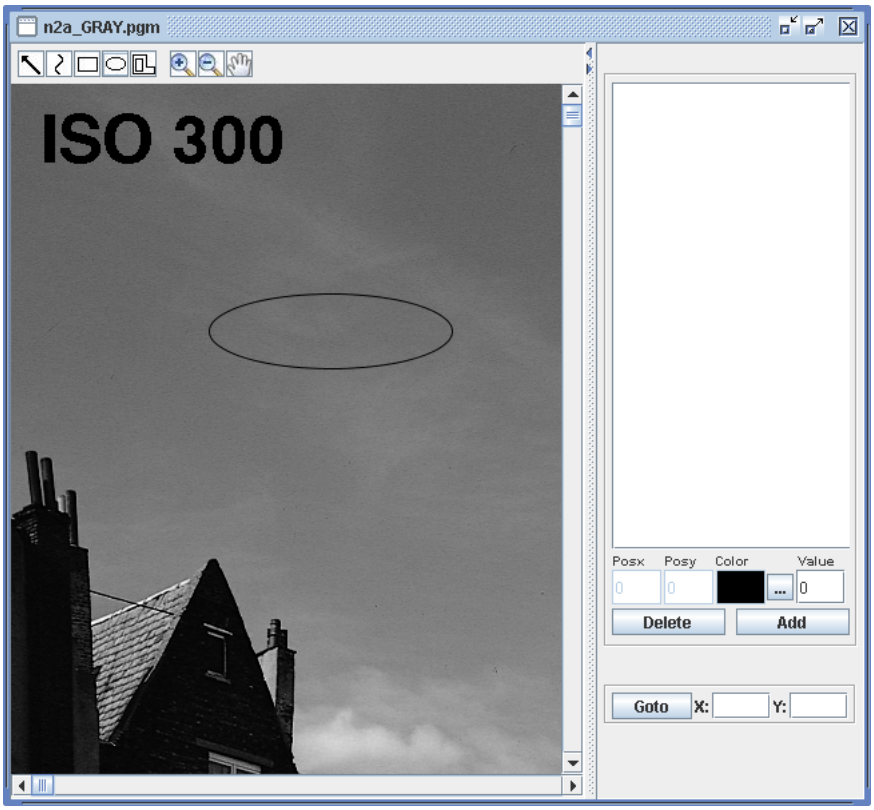


Figura 4.8: El·lipse generada sobre una imatge PGM.

4.4.4 Figures compostes

Amb la classe **ComposedShape** controlarem el dibuix de figures compostes. Aquest és un tipus de figures que pot tenir altres figures a dins. A diferència de la resta de figures vistes fins ara, quan premem de nou el mouse sobre la imatge no crearem una figura complexe nova sinó que associarem una nova figura a la figura anterior. A la classe tenim els següents atributs definits:

- *path* (del tipus *GeneralPath*): on anirem afegint les diferents figures dibuixades, enllaçades unes amb les altres.
- *color* (del tipus *Color*): hi guardarem el color amb el que hem dibuixat la figura.
- *ptsCompShape* (del tipus *Vector*): hi anirem afegint els punts que formen la figura.

Des de la classe **ImageVisor** controlem a l'event *mousePressed* quan hem de crear una figura composta, en funció de l'acció activa en aquell moment, a partir d'aquí cada cop que pitgem sobre la imatge crearem un nou *SimpleShape* associant-lo a l'anterior. Veiem que el GeneralPath el creem amb l'atribut *WIND_EVEN_ODD*, per tal de poder calcular

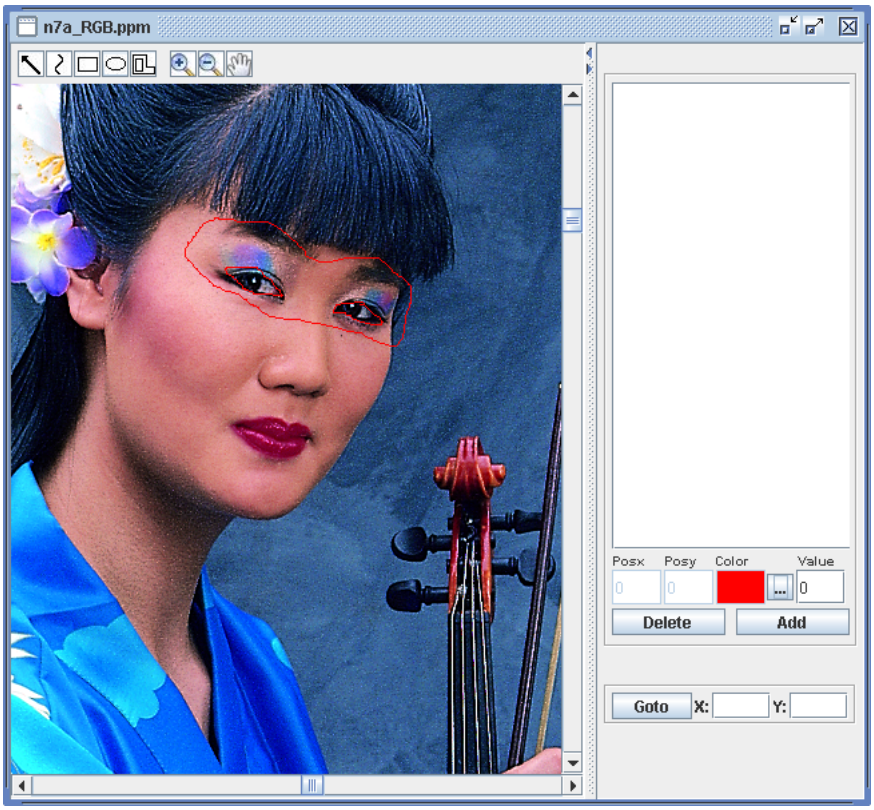


Figura 4.9: Figura composta generada sobre una imatge PPM.

correctament quins són els punts que queden a l'interior de la figura, quan haguem de crear la regió.

L'altre situació a controlar és a l'hora de tancar la regió. Sempre cridem als mètodes: *SimpleShape.closeShape* i *ComposedShape.closeComposedShape*, perquè no sabem si l'usuari continuarà dibuixant o no. En el mètode *ComposedShape.closeComposedShape* el que fem és recórrer les figures creades, afegint-les al *ComposedShape* mitjançant el mètode *append*.

4.4.5 Generant les regions

Quan convertim la figura en regió, creem un objecte del tipus **BitMap**. El que fem amb aquesta classe és associar-li el rectangle més petit possible que inclogui la figura dibuixada, indicant quins punts d'aquest rectangle estan dins de la figura i quins estan fora. Per tal de reduir el cost en memòria, ho calculem mitjançant un atribut de tipus byte[]. El creem de tamany (amplada*alçada) i usem cada bit per indicar si està a dins o fora (1 = està a dins, 0 = està a fora). A la figura següent en podem veure una representació. A la classe

GenRegionFrame tenim l'atribut *lBitMap* del tipus *LinkedList* on anirem afegint totes les regions creades.

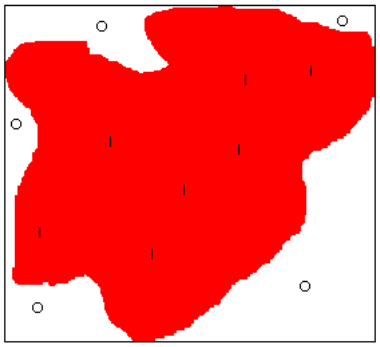


Figura 4.10: Valors de la regió guardada a BitMap.

A la classe **BitMap** disposem dels següents atributs:

- id: de tipus *int* usat com a identificador de la regió.
- xyLeft, de tipus *int* usat per a guardar el punt (x,y) situat més a l'esquerra.
- maxX, de tipus *int* usat per a guardar el punt X situat més a la dreta.
- maxY, de tipus *int* usat per a guardar el punt Y situat més a la dreta.
- minX, de tipus *int* usat per a guardar el punt X situat més a l'esquerra.
- minY, de tipus *int* usat per a guardar el punt Y situat més a l'esquerra.
- value, de tipus *int* usat per a guardar el valor amb el qual guardarem la regió com a màscara.
- bitMap, de tipus *byte[]*, usat per a guardar el rectangle que inclou a la regió. Posarem un 0 al que queda fora de la frontera de la regió i un 1 al que queda dins de la regió.
- color, de tipus *Color*, usat per a guardar el color que defineix a la regió.
- shape, de tipus *Shape* i usat per emmagatzemar la figura dibuixada. Usem aquest atribut posteriorment per omplir la figura amb el mateix color.

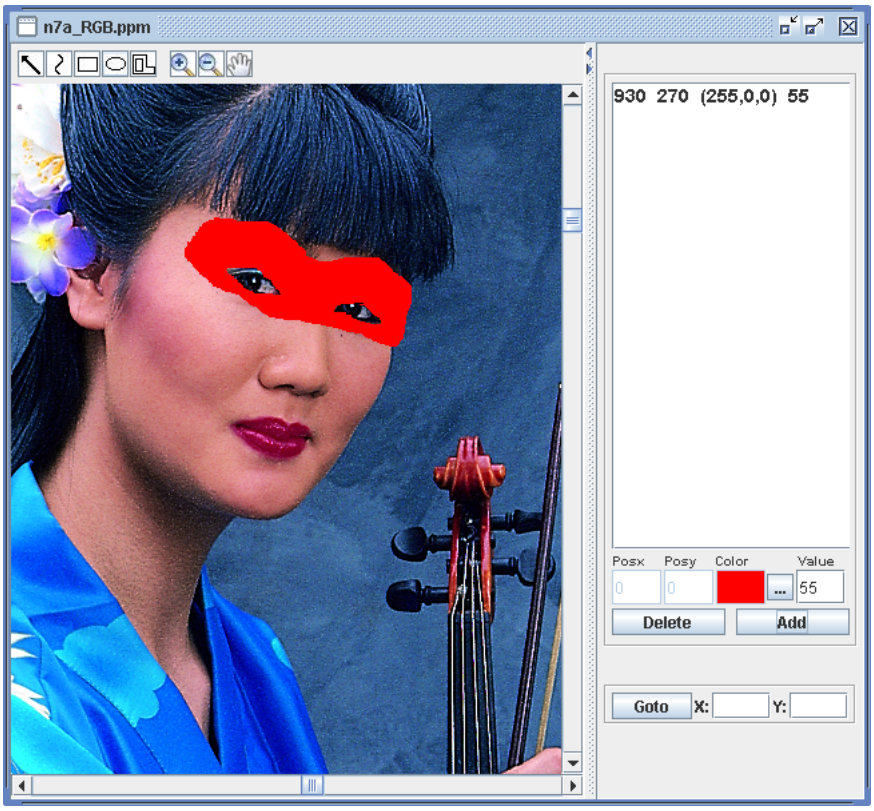


Figura 4.11: Regió generada a partir d'una figura composta.

El procés seguit per a crear un objecte *BitMap* és el següent: primer obtenim la posició (x,y) màxima i mínima. A continuació creem un objecte *BitMap* a partir d'aquests valors, obtenint el rectangle que engloba la figura. Assignem el color i el valor. Tot seguit omplim l'array de bits posant un 1 si el píxel en qüestió cau a dins de la figura i un 0 en cas contrari. Finalment esborrem la figura (objecte *shape*). A partir d'ara és una regió i al dibuixar-la ho farem pintant el seu interior, cridant al mètode *fill* del context gràfic passant el *shape* com a atribut.

4.5 Configuració de l'aplicació

Hi ha diferents opcions de configuració que hem anat afegint a l'aplicació per tal de fer-la més amigable de cara a l'usuari i facilitar-li el treball. Són opcions que han anat sorgint a mida que anàvem avançant en el projecte. En els propers apartats explicarem quin és l'origen d'aquesta necessitat i com l'hem solucionat.

4.5.1 Mapa de píxels

Hem vist que podem realitzar diferents tipus de figures ja sigui sense zoom aplicat o bé amb un determinat nivell de zoom aplicat. En cas que estiguem treballant amb zoom positiu, els píxels reals de la imatge no es correspondran amb els píxels que veiem per pantalla. A la imatge que veiem per pantalla un píxel real es correspondrà amb x píxels. I les figures que dibuixem s'han de correspondre amb els píxels reals de la imatge. En funció dels colors o dels grisos existents, ens pot resultar fàcil diferenciar els píxels reals, però moltes vegades no sabrem diferenciar-ho.

Per tal de solucionar aquest problema, hem implementat el que anomenem un mapa de píxels, que no es més que dibuixar sobre la imatge una graella que diferenciï els píxels reals de l'aplicació amb el color que nosaltres haguem seleccionat prèviament.

Aquesta opció es troba implementada a la classe *PixelMapConf*. Al seleccionar l'opció se'ns mostra un panell amb un botó per a la selecció del color i un *textfield* on mostrarem el color seleccionat. Quan pitgem el botó, se'ns obre *JColorChooser* que ens permet seleccionar el color de tot el ventall possible. Si pitgem el botó *Accept*, des d'aquesta mateixa classe cridarem al mètode *displayImage* passant com a paràmetres la imatge i el color seleccionat.
En el mètode *paintComponent* de la classe *ImageVisor* dibuixarem la graella sempre i quan l'atribut *pixelMap* sigui diferent de *null*. A la figura 4.4.2 podem veure l'efecte produït a l'activar el Mapa de píxels. Cada quadrat representa un píxel real de la imatge. La graella només la visualitzarem si estem aplicant un nivell de zoom positiu. En cas contrari no té

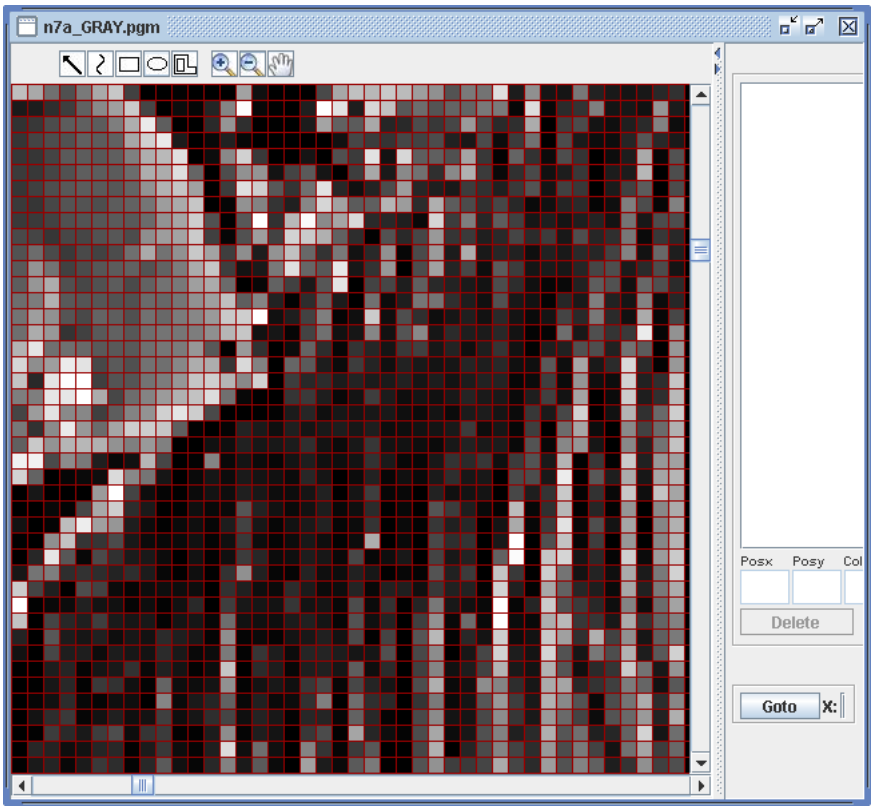


Figura 4.12: Mapa de píxels activat per a una imatge PGM.

sentit.

4.5.2 Visor global

La finestra corresponent al visor global és independent de la resta de finestres. I, per tant, la podem tancar independentment de la resta de finestres. Inicialment, quan la tancàvem, no la podíem tornar a obrir. Per poder tornar a visualitzar el visor, havíem de tancar la finestra corresponent a la imatge i tornar-la a obrir.

És per solucionar aquest aspecte que hem afegit una opció de menú (GenRegion-View-GlobalImage) que ens permet tornar a obrir la finestra del visor global corresponent a la imatge que tenim actualment carregada en qualsevol moment.

4.5.3 Escala del visor

Les imatges que obrirem no tenen totes el mateix tamany. I el que pot ser una escala correcte per un determinat tamany pot resultar massa petita o massa gran per altres tamanys

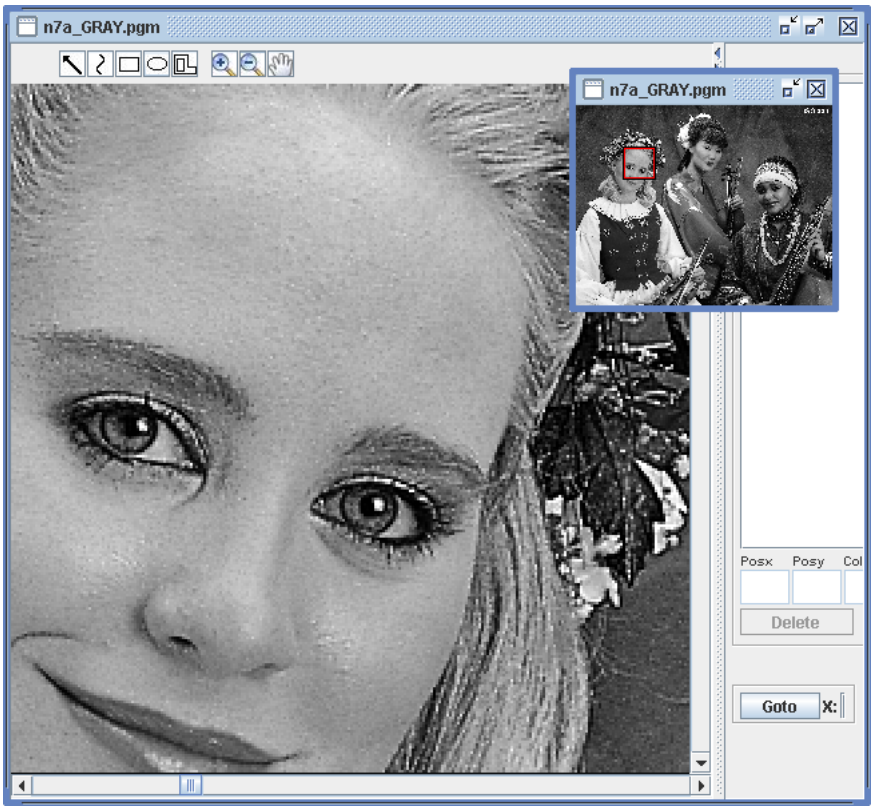


Figura 4.13: Imatge amb el seu visor global.

d'imatges. Per tal de corregir-ho, hem afegit una opció mitjançant un menú a la pantalla principal (GenRegion-Options-GlobalVisorConfiguration) que mitjançant un *Slider* permet seleccionar l'escala que volem aplicar al visor. Cal tenir en compte que perquè s'apliqui la nova escala seleccionada, haurem de tancar i tornar a obrir el visor global.

A la classe *GlobalVisorConf* és on gestionem la configuració de l'escala del visor. És una interfície composada per un *JFrame*, un *JPanel*, el *JSlider* per seleccionar l'escala que volem i dos *JButton* per acceptar o cancel·lar. Per defecte l'escala és sempre 1/10. Un cop hem modificat el valor i acceptem, assignem el valor del *JSlider* a l'atribut *GenRegion.gVisorScale*.

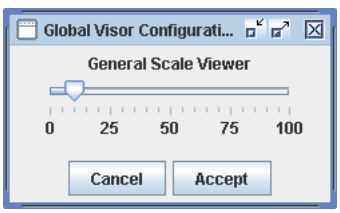


Figura 4.14: Pantalla per a la configuració del visor global.

4.5.4 Opcions d'emmagatzematge

Com a requeriments de l'aplicatiu teníem únicament la generació d'una màscara de les regions generades. Però a l'hora de implementar les regions, vam trobar interessant també generar una imatge tal i com la veiem per pantalla. Per tal de poder triar quina de les dues opcions volem a l'hora de desar la imatge, hem afegit aquesta opció de configuració *GenRegion-Options-Save File Options* implementada a la classe *SaveFileConf*.

És una interfície formada per un *JFrame*, un *JPanel*, dos *JRadioButton* (un per a cada opció) i dos *JButton*. Quan acceptem, assignem el valor 0 o 1 en funció de l'opció triada a *GenRegion.GSaveOption*. Quan des de la pantalla principal pitgem el botó *Save File* generarem la màscara o bé la imatge en funció del que haguem seleccionat prèviament en aquesta pantalla. Per defecte sempre es gravarà la màscara.

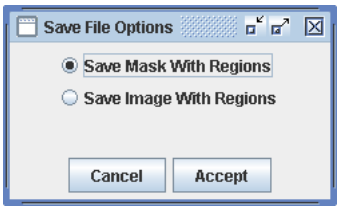


Figura 4.15: Pantalla per a la configuració de l'opció d'emmagatzematge.

Capítol 5

Resultats

A continuació exposarem les proves efectuades a l'aplicatiu. Hem realitzat proves a nivell de plataforma i a nivell de temps de càrrega de les imatges en comparació amb una aplicació de referència.

5.1 Resultats obtinguts

Hem comprovat, en primer lloc, que l'aplicació es pugui executar en diferents plataformes obtenint en cada cas la mateixa funcionalitat. Ho hem provat sobre les següents plataformes:

- **Windows**
  - **Windows XP Home Edition 2002 Version, ServicePack2** i amb el següent maquinari: processador Intel Core Duo a 1,83 GHz, 1 Gb de memòria RAM i disc dur de 88 Gb (75 Gb d'espai lliure).
  - **Windows XP Professional** i amb el següent maquinari: processador Pentium 4 a 2.8 GHz, 512 Mb de memòria RAM i disc dur de 112 Gb (20 Gb d'espai lliure).
- **Linux**
  - **Linux Ubuntu 6.0.6** amb el següent maquinari: processador Pentium 4 a 2,8 GHz, 512 Mb de memòria RAM i disc dur de 80 Gb (21 Gb d'espai lliure).

  - **Linux Debian 4.0** amb el següent maquinari: processador Pentium 4 a 3 GHz, 1Gb de memòria RAM, disc dur de 160 Gb (55 Gb d'espai lliure).

- **Mac**

  - **MacOS Tiger 10.4** amb el següent maquinari: processador PowerPC a 2 Ghz, 1 Gb de memòria RAM, disc dur de 250 Gb (100 Gb d'espai lliure).

Hem comparat la memòria consumida per l'aplicació amb la consumida pel programa *Gimp* a l'hora d'obrir la mateixa imatge. A l'hora d'el·laborar el banc de proves, s'han seleccionat imatges dels dos formats possibles (PPM i PGM) i de diferents tamanys.
Com podem veure a la següent taula, el consum de memòria efectuat per l'aplicació és força menor que el realitzat per *Gimp*. Això és degut a que no carreguem tota la imatge sinó únicament aquella part de la imatge que podem visualitzar per pantalla. Hem fet les proves mostrant el mateix tamany de la imatge amb totes dues aplicacions (412*517 píxels). I llegint el 100% dels píxels, donat que d'entrada *Gimp* no carrega el 100% dels píxels.

	Nom Imatge	**Tamany imatge**	***Gimp***	***GenRegion***
img01	n1a_GRAY.pgm	2.881 Kb	43.526 Kb	36.632 Kb
img02	n5a_GRAY.pgm	2.881 Kb	43.689 Kb	35.320 Kb
img03	n1_GRAY.pgm	5.121 Kb	52.576 Kb	35.356 Kb
img04	n2_GRAY.pgm	5.121 Kb	52.712 Kb	36.272 Kb
img05	n5a_RGB.pgm	8.641 Kb	55.200 Kb	36.660 Kb
img06	n1_RGB.ppm	15.261 Kb	71.940 Kb	38.352 Kb
img07	n2_RGB.pgm	15.361 Kb	73.220 Kb	38.100 Kb
img08	p1_GRAY_07830.pgm	35.157 Kb	169.040 Kb	48.316 Kb
img09	p1_GRAY_07830.pgm	97.657 Kb	198.428 Kb	69.616 Kb

Si ens fixem en les dades, a mida que creix el tamany de la imatge també creix el consum de memòria de l'aplicació, això és degut a què les proves s'han realitzat amb càrrega complerta. És a dir, carregàvem la imatge i també el visor global. Podem reduir encara

més el consum de memòria de l'aplicació GenRegion reduint l'escala del visor global o fent que no aparegui (escala = 0). A la següent gràfica podem veure clarament que, mentre que el consum de memòria de GenRegion es manté força estable, el de l'aplicació *Gimp* és directament proporcional al tamany de la imatge llegida.

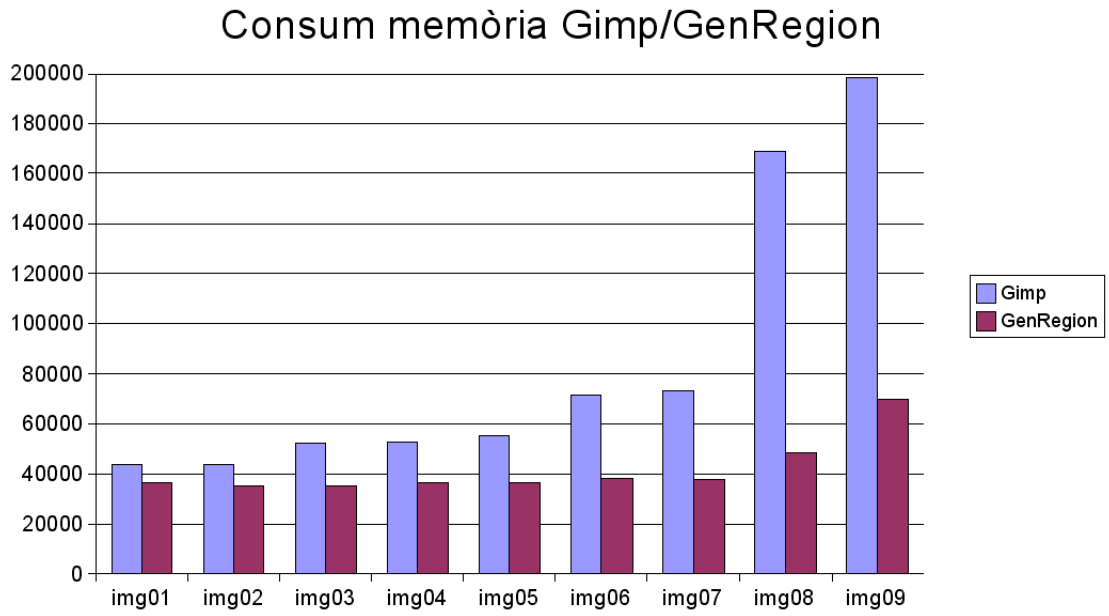


Figura 5.1: Comparació del consum entre ambdues aplicacions.

Capítol 6

Conclusions

En aquest capítol, realitzarem una valoració global de la feina feta amb un repàs a tots els coneixements que hem adquirit per portar a bon terme aquest treball. Tot seguit abordarem futures línies d'acció respecte a la millora d'aquest projecte.

6.1 Valoració global del projecte

Per finalitzar el document, exposarem les conclusions de l'el·laboració del projecte. Vam començar exposant la necessitat d'una aplicació que permetés generar regions sobre imatges PGM a l'hora que es pogués consultar i accedir fàcilment a la llista de regions generades. Vam veure que un requeriment molt important era el consum de memòria, donat que l'obertura de grans imatges podia provocar que la màquina virtual de Java no fos capaç de gestionar-ho. Normalment, quan s'optimitza el consum de memòria provoca una penalització en el temps de l'operació. Hem procurat en tot moment que el consum extra en temps no esdevingui un obstacle, aconseguint que els càlculs no consumeixin una gran quantitat de temps.

Seguidament hem vist quina és la solució que finalment hem adoptat, explicant el perquè de cada decisió i quines ampliacions hem fet. Hem obtingut una aplicació que ens permet l'obertura d'imatges PGM i PPM i dibuixar-hi regions d'interès amb una interfície que ens permet maximitzar la superfície de treball de la imatge alhora que un ràpid accés a les

diferents regions creades, disposant en tot moment de informació dels seus valors i de la seva ubicació.

Hem detallat les proves realitzades sobre l'aplicació per tal de poder determinar que s'han assolit els objectius fixats inicialment. Podem dir, doncs, com a conclusió, que s'han acomplert els requeriments inicials i l'aplicació soluciona els requeriments tant funcionals com no funcionals establerts inicialment.

6.2 Possibles millores i línies de continuació

Hem vist a l'apartat anterior que s'han assolit els objectius inicials, però també durant el procés d'implementació han anat sorgint noves necessitats i s'han anat observant possibles millores per a l'aplicació. Algunes d'aquests nous punts s'han pogut assolir, com són: mapa de bits, *hints* descriptius, diferents opcions d'emmagatzematge o la configuració del visor. D'altres, però, no han pogut ser implementats. A continuació detallarem aquells punts que es podrien millorar així com línies de futur de l'aplicació.

- **Optimització lectura imatges:** Ara mateix cada vegada que fem un desplaçament de la imatge, tornem a llegir tota la matriu que es mostrarà per pantalla, l'alçada i amplada que mostrarem des de la nova posició. Tot i que des del punt de vista de l'usuari, aquest consum extra és gairebé inapreciable, es podria disminuir els accessos a fitxer, llegint només aquella part que no tenim a la matriu de treball.

- **Lectura de nous formats d'imatges:** Hem trobat que fóra molt interessant poder obrir altres formats d'imatges, com són JPEG, GIF, RAW i JPEG2000. S'hauria de centralitzar tot en un mètode de lectura que retornés una matriu dels punts llegits. Els diferents punts d'accés a una nova lectura de la imatge (*scroll*, *panning*, zoom...) cridarien a aquest mètode des d'on es faria la lectura de la imatge en funció del format.

- **Obrir més d'una imatge al mateix temps:** L'estructura actual de l'aplicació amb les tres pantalles (pantalla principal, pantalla de treball i visor global) facilita

la futura obertura de més d'una imatge. Així des de la pantalla principal podríem obrir una nova imatge sense haver de tancar l'anterior. Per implementar-ho, hauríem de gestionar una *array* de la classe *GenRegionFrame* i controlar sempre sobre quina imatge estem treballant.

- **Implementar el *roll* del mouse:** La majoria d'aquests dispositius incorpora una rodeta en el centre del dispositiu que permet avançar endavant o enrera molt ràpidament. Ens permetria fer un *scroll* de la imatge molt ràpidament i també es podria implementar per fer zoom sobre la imatge, en funció del botó que haguéssim pitjat.

- **Poder dibuixar polígons:** Disposar d'una nova opció per al dibuix de figures, que permeti crear polígons, línies rectes que enllacin els diferents punts pitjats pel *mouse*.

- **Millora del *ComposedShape*:** Poder generar figures compostes amb els altres tipus de figures. Ara mateix quan dibuixem una figura de tipus composta només tenim una opció de dibuix que és la de llapis o "mà alçada". Seria interessant, a l'hora de crear figures compostes, poder intercalar dibuixos realitzats amb el llapis amb rectangles o amb el·lipses.

- **Variar el gruix del punter a l'hora de dibuixar:** Ara mateix es dibuixa sempre amb un únic tamany sigui quin sigui el nivell de zoom aplicat. Seria interessant poder variar aquest tamany per facilitar la visualització del dibuix fet.

- **Doble clic sobre les regions ens posicioni a la seva ubicació**: ens permetria guanyar velocitat a l'hora de treballar que al clicar sobre qualsevol de les regions existents a la llista, ens posicionés automàticament a la seva ubicació a la imatge.

Bibliografia

[ANH] Ant homepage

URL:http://ant.apache.org

[DBU] Double Buffer

URL:http://www.codeproject.com/java/javadoublebuffer.asp

[HOR00] Horton, Ivor: *Beginning Java 2*,Wrox Press, Ltd. 2000

[JAP] Java API: http://java.sun.com/j2se/1.4.2/docs/api

[JPA] Java painting

URL:http://java.sun.com/products/jfc/tsc/articles/painting/index.html

[JSP] Java SplitPanes

URL:http://java.sun.com/docs/books/tutorial/uiswing/components/splitpane.html

[JTO] Java ToolTips

URL:http://www.java2s.com/Code/Java/Swing-JFC/ToolTipLocationExample.htm

[ROD01] Rodrigues, Lawrence H.: *Building Imaging Applications with Java Technology*, Addison Wesley 2001

[PGM] PGM Format

URL:http://netpbm.sourceforge.net/doc/pgm.html

[PPM] PPM Format

URL:http://netpbm.sourceforge.net/doc/ppm.html

[WAL04] Walrath, Kathy: *The JFC Swing Tutorial Second Edition*, Sun Microsystems 2004

Apèndixs

Apèndix A

Manual d'Usuari de l'aplicació GenRegion

A.1 Introducció

GenRegion és una aplicació desenvolupada amb Java que permet generar regions d'interès sobre imatges PGM o PPM amb un mínim cost de memòria. A l'hora d'obrir les imatges, només carrega en memòria aquella part de la imatge que es pot mostrar per pantalla, reduint així el consum de memòria RAM i permetent l'obertura d'imatges molt grans.

L'aplicació consta de tres pantalles diferents. Una primera pantalla (pantalla principal) des d'on seleccionem la imatge que volem obrir i des d'on podem accedir a les diferents opcions de configuració i dues pantalles més que apareixeran a l'obrir una imatge: la pantalla de treball, on visualitzem la imatge carregada sense escalat ni compressió i des d'on podem generar les regions d'interès, i una segona pantalla consistent en un visor global de tota la imatge a una escala reduïda.

Un cop oberta la pantalla de treball, podrem fer diferents operacions sobre la imatge com *zoom out*, *zoom in*, *panning*. Així com dibuixar les regions d'interès usant per a tal efecte quatre opcions possibles: dibuix lliure (o dibuix a "mà alçada"), rectangle, el·lipse i regions complexes (són regions que dins tenen altres regions). Cal destacar que el dibuix

de les regions el podrem fer independentment del nivell de zoom aplicat. Quan canviem el nivell de zoom, les regions dibuixades es redimensionaran al nou tamany.

El visor global ens permet tenir una vista de tota la imatge i desplaçar-nos ràpidament per ella usant el marc de visualització. El visor global el podrem obrir i tancar sense afectar a la imatge carregada en memòria.

A.2 Execució

GenRegion consta d'una sola aplicació encapsulada en el jar GenRegion.jar que podem trobar al directori /dist. Per executar l'aplicatiu l'única cosa que hem de fer és executar aquest jar. Cal tenir en compte que GenRegion és multiplataforma i, per tant, el podrem executar en qualsevol entorn, ja sigui *Windows*, *Linux*, o qualsevol sistema operatiu que tingui instal·lada la màquina virtual Java. Al executar-lo ens apareix una primera pantalla des d'on es centralitzen les diferents opcions de configuració, i des d'on seleccionarem la imatge amb la que volem treballar.

A.3 Descripció de l'espai de treball

A continuació descriurem la funcionalitat de les diferents pantalles que ens trobem a l'aplicatiu.

A.3.1 Pantalla principal

És la pantalla que ens apareix tot just executem l'aplicació. Des d'aquesta pantalla és des d'on seleccionarem la imatge que volem obrir i des d'on indicarem on la volem desar un cop generades les regions d'interès. Les opcions de configuració es troben centralitzades en aquesta pantalla a través de les opcions del menú.

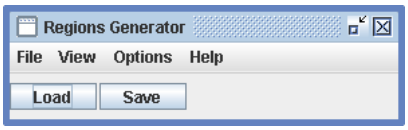


Figura A.1: Pantalla principal.

Com veiem a la Figura A.1, consta de dos botons i de quatre opcions de menú (File, View, Options i Help).

Carregar una imatge

Per carregar una imatge, tenim dues vies d'accés, pitjant el botó *Load* o bé accedint a l'opció de menú *File - Load*. Al prémer qualsevol d'aquestes dues opcions, ens apareixerà una pantalla per tal de seleccionar la imatge que volem obrir. La imatge ha de tenir extensió PGM o bé PPM. Si intentem obrir qualsevol altre format ens apareixerà un missatge d'error

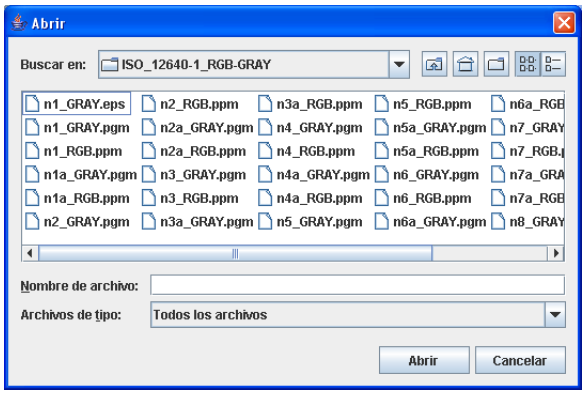


Figura A.2: Selecció de fitxers (imatges).

per pantalla. Un cop oberta la imatge seleccionada ens apareixeran dues noves pantalles. La pantalla de treball i el visor global. Tant a la finestra de treball com a la finestra del visor global apareixerà el nom del fitxer obert a la part superior de la pantalla.

Desar una imatge

Per desar una imatge, tenim també dues vies d'accés, pitjant el botó *Save* o bé accedint a l'opció de menú *File - Save*. Al prémer qualsevol d'aquestes dues opcions, ens apareixerà una pantalla per tal d'indicar el nom amb el qual volem desar la imatge. L'extensió ha de ser PGM o bé PPM, però cal tenir en compte que si tenim configurat com a opció d'emmagatzematge *save as a mask* l'extensió haurà de ser obligatòriament PGM. El nom del fitxer ha de ser també diferent de l'original, per tal d'evitar perdre la imatge inicial.

Sortir

Per sortir de l'aplicació i tancar totes les pantalles que tinguem obertes tenim dues opcions, o bé prémer el botó amb una creu que es troba situat a la cantonada superior dreta de la pantalla principal o bé triar l'opció del menú *File - Exit*.

Activar mapa de píxels

El mapa de píxels ens mostrarà una graella sobre la imatge principal ajudant-nos a identificar clarament tots els píxels reals de l'aplicació. Aquesta opció només té sentit en el cas que apliquem un nivell de zoom positiu sobre la imatge. Aplicant un nivell de zoom positiu, un píxel real de la imatge passa a ocupar x píxels per pantalla, tots del mateix color. La graella ens ajuda a visualitzar clarament aquests píxels. Per activar el Mapa de píxels, hem d'accedir a l'opció del menú *View - PixelMap*, ens apareixerà una pantalla tal com ens mostra la figura A.3, on haurem d'activar el *checkbox*, prémer el botó *Choose color* per seleccionar el color amb el que volem que es mostri la graella i a continuació prémer el botó *Accept*. Cal tenir en compte que aquesta opció la podem activar abans de carregar la imatge o bé un cop ja ha estat carregada.

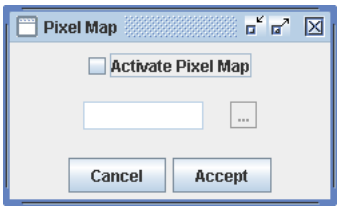


Figura A.3: Activar mapa de píxels.

Mostrar Visor Global

Quan obrim una imatge, automàticament es mostra també el visor global. Però donat que podem tancar independentment el visor sense tancar la pantalla de la imatge associada, disposem d'aquesta opció per tornar a carregar el visor. El visor només es mostrarà si tenim

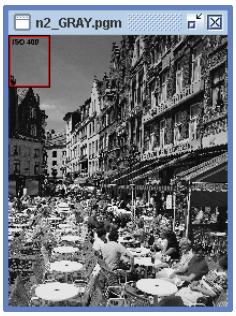


Figura A.4: Visor global.

una imatge carregada, en cas contrari no es mostrarà el visor. Per tornar a visualitzar el visor hem d'accedir a l'opció del menú *View - Global Visor*.

Configuració del Visor Global

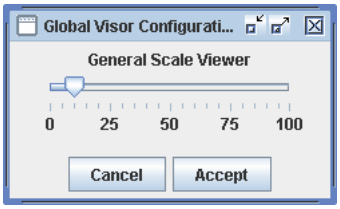


Figura A.5: Configuració de l'escala del visor.

Accedint a l'opció del menú *Options - Global Visor Configuration*, ens apareix una pantalla que ens permet indicar la escala que volem aplicar al visor global. Per defecte, l'escala és 1/10. Canviem l'escala a aplicar desplaçant el *slider* a l'opció que ens interessa. Si tenim el visor global obert, l'haurem de tancar per tal que els canvis tinguin efecte.

Opcions d'emmagatzematge

Accedint a l'opció del menú *Options* - *Save File Options* podem indicar quina és l'opció que volem a l'hora de desar la imatge.
Hi ha dues possibilitats:

- *Save Mask with Regions*: amb aquesta opció desarem una màscara amb les regions triades. És a dir, es guardarà una imatge del mateix tamany que la imatge inicial, però tota ella en negre. Només apareixeran, ressaltades en un altre color, les regions dibuixades, que apareixeran amb el valor que se'ls hi hagi assignat. Si triem aquesta opció, quan desem la imatge l'hem de desar obligatòriament en format PGM. En cas contrari ens apareixerà un error.

- *Save Image with Regions*: amb aquesta opció desarem la imatge tal qual la veiem per pantalla, amb les regions dibuixades amb el color triat. És a dir, tindrem una imatge igual a la inicial però amb les regions afegides amb el color associat.

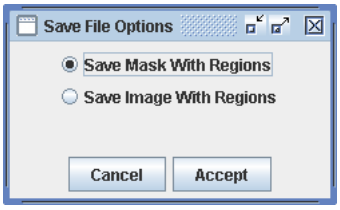


Figura A.6: Opcions d'emmagatzematge.

A.3.2 Marc de treball

En aquesta pantalla és on treballarem amb l'aplicació, generant i emmagatzemant les regions. La pantalla es troba dividida en dues parts mitjançant un *split panel*. Aquesta configuració ens permet guanyar espai per a la imatge i accedir ràpidament a la informació de les regions generades. Al panell de l'esquerra hi trobem una botonera a la zona superior amb els següents botons: *clean*, *draw*, *rectangle*, *el·lipse*, *complex shape*, *zoom in*, *zoom out* i *panning*. A sota de la botonera apareix un panell on es mostrarà la imatge carregada.
Al panell de la dreta trobem la llista de les regions amb els botons per afegir-ne i eliminar-ne, i una utilitat per situar-nos en la posició de la imatge que vulguem.

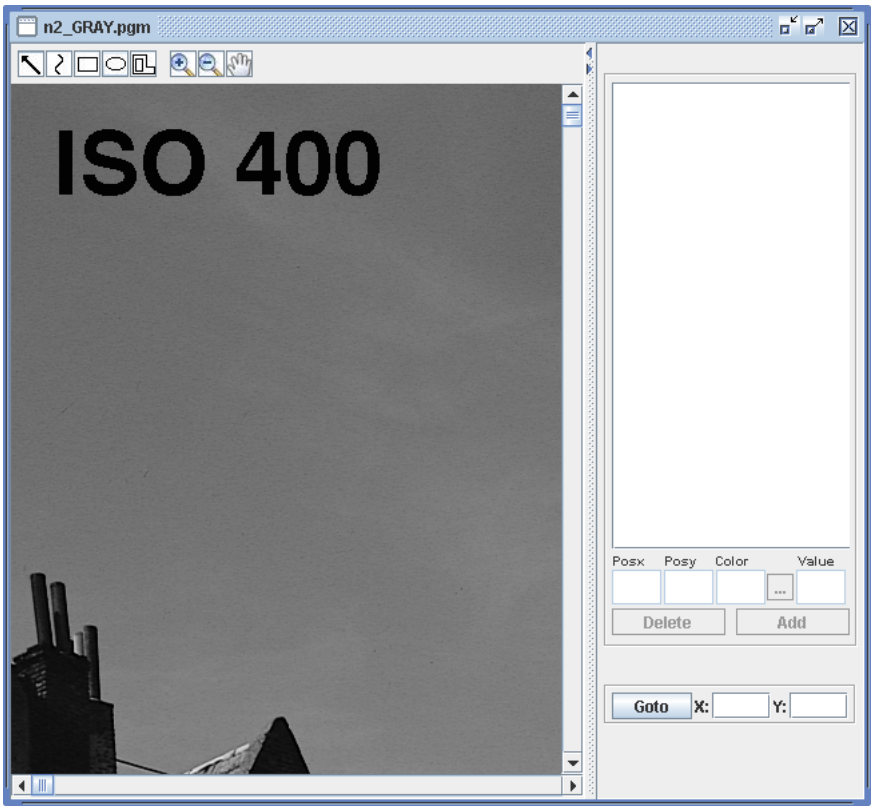


Figura A.7: Marc de treball.

Cal tenir en compte també, que aquesta pantalla és redimensionable i en podem modificar el tamany.

Clean

Aquest botó ens permet eliminar la selecció que haguem fet anteriorment. D'aquesta forma si ens situem amb el *mouse* sobre la imatge podrem tornar a visualitzar la informació dels píxels sobre els que ens situem (apareix un *hint* informant-nos de la posició i el color del píxel).

Draw

Pitjant aquest botó, s'activa la selecció de color per a dibuixar una regió que per defecte és la del color negre. Un cop triat el color, ja podrem dibuixar sobre la imatge mantenint pitjat el botó esquerra del *mouse*. Si ens desplacem massa a prop de les cantonades de la imatge, automàticament la imatge es desplaçarà per permetre'ns continuar dibuixant. Quan deixem de pitjar el botó esquerra del mouse, automàticament es tancarà la regió

creada, unint el primer amb el darrer punt del dibuix.

Rectangle

Prement aquest botó disposem de la mateixa funcionalitat que en el punt anterior, però enlloc de permetre'ns realitzar un dibuix lliure, dibuixa un rectangle en funció del desplaçament que fem amb el *mouse*.

Ellipse

Prement aquest botó disposem de la mateixa funcionalitat que en el punt anterior però dibuixant una el·lipse en funció del desplaçament efectuat amb el *mouse*.

Cal tenir present que a l'hora de dibuixar amb les opcions vistes fins ara, un cop es tanqui la regió, si tornem a dibuixar es crearà una altre figura i es perdrà l'anterior. Això és així per permetre dibuixar una altre figura ràpidament sense haver de perdre temps esborrant l'anterior. Si un cop dibuixada una figura la volem mantenir haurem de prémer el botó *"Add Region"*. En cas contrari, el que hem de fer és tornar a dibuixar fins obtenir la figura desitjada.

Complex Region

Amb aquesta opció se'ns permet dibuixar una regió complexe. És una regió que dins pot tenir altres regions. Podem obtenir regions en forma d'anella. En aquest cas quan acabem de dibuixar una regió, es tanca automàticament però quan tornem a dibuixar no perdem l'anterior sinó que hi afegim una nova regió. En podem veure un exemple a la figura A.8. Veiem que la regió en color blau cel és una regió que dins conté dues regions més. Quan l'hem passat a regió, s'han pintat els punts existents entre una i altre frontera.
A la figura A.8 podem veure un exemple del resultat d'haver creat una regió amb cada opció. Amb color groc una regió creada amb l'opció *Draw*, amb color lila una regió creada a partir d'un rectangle, amb color blau marí una regió creada a partir d'una el·lipse i amb color blau cel una regió creada a partir d'una figura composta.

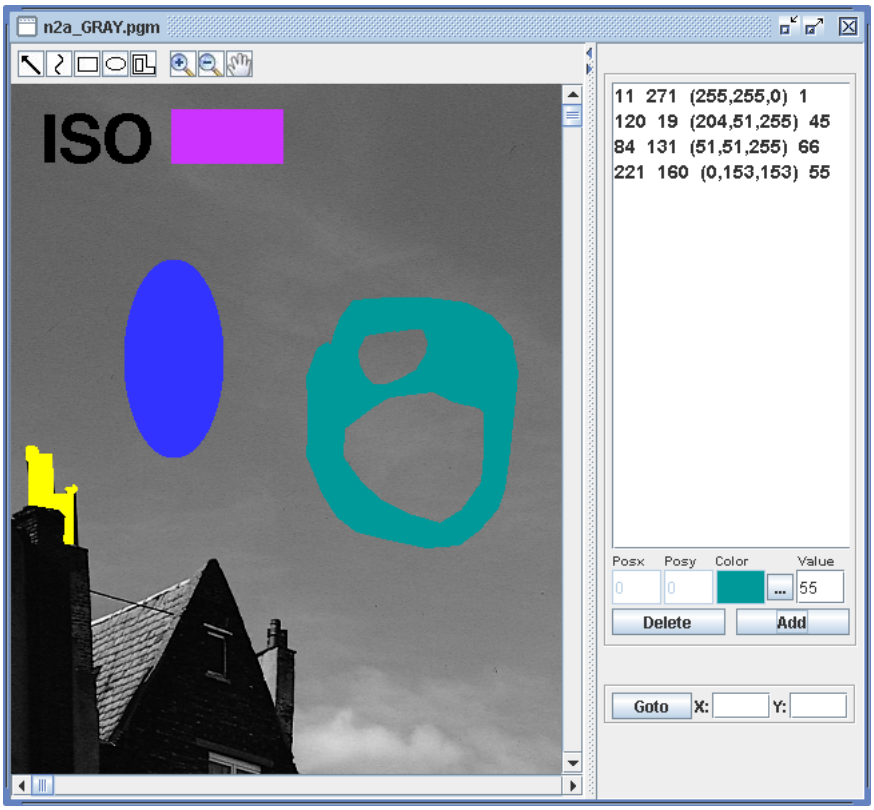


Figura A.8: Regions creades.

Zoom in

Amb aquesta opció apliquem un nivell de zoom positiu, és a dir, fem la imatge més gran. L'algoritme que es segueix per implementar el zoom és duplicar els píxels multiplicant cada cop per dos el tamany inicial de la imatge. Hem de tenir en compte que tot i tenir zoom aplicat podem dibuixar regions igualment no afectant al tamany final de la regió. Amb un nivell de zoom positiu aplicat *(zoom in)* podem activar el mapa de píxels per veure clarament els píxels originals.

Zoom out

Amb aquesta opció apliquem un nivell de zoom negatiu, és a dir, fem la imatge més petita. L'algoritme que es segueix per implementar el zoom és reduir el nombre de píxels dividint per dos el tamany inicial de la imatge cada vegada que apliquem el zoom.

Tant per al *zoom in* com per al *zoom out*, les regions que tinguem dibuixades es redimensionaran en funció del nou tamany de la imatge.

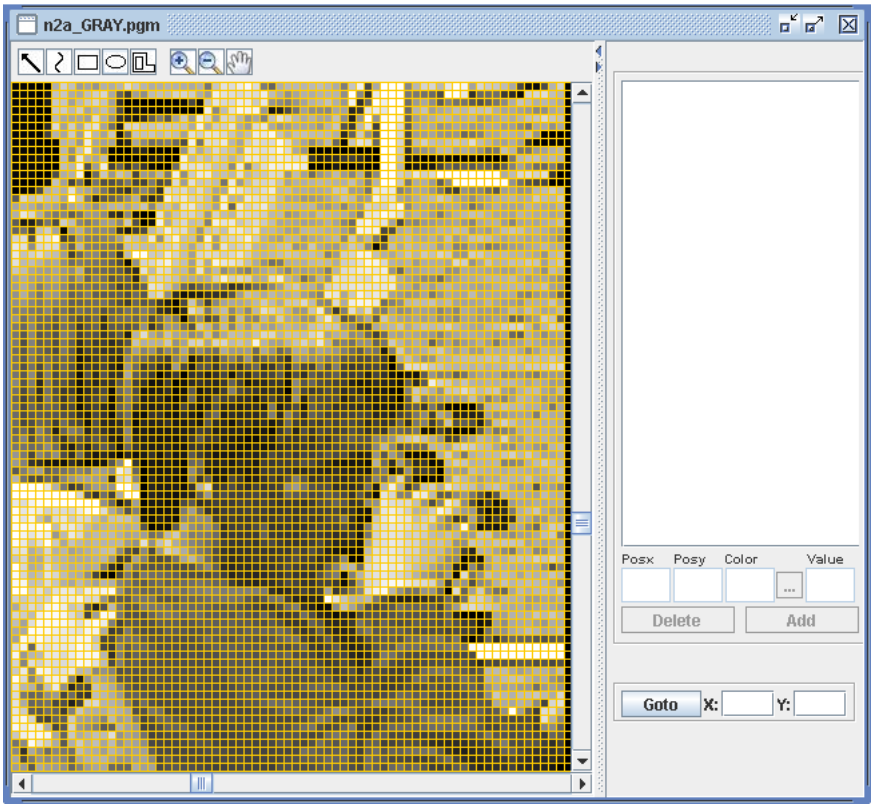


Figura A.9: Imatge amb zoom i mapa de píxels activats.

Panning

Amb aquesta opció podem moure la imatge pitjant amb el botó esquerra sobre la imatge i desplaçant-la cap allà on vulguem. Veurem que al pitjar aquest botó, el punter del *mouse* es converteix en una mà.

Botó Add region

Un cop haguem dibuixat una figura amb qualsevol de les opcions vistes anteriorment i vulguem desar-la, pitjarem aquest botó, calculant tots els punts que formen part de la regió i afegint la regió dibuixada a la llista de regions. Abans d'afegir la regió haurem de triar el valor (de 1 a 255) amb que volem desar-la. Un cop afegida la regió, aquesta s'afegirà també, a l'escala corresponent, al visor global.

Botó Delete region

Amb aquest botó podrem esborrar una regió de la llista de regions, eliminant-la també del visor global.

A.3.3 Visor Global

Amb el visor global disposem d'una vista a escala de tota la imatge. Tenim un marc que es correspon amb el que visualitzem de la imatge i que podem desplaçar per moure'ns ràpidament a través de la imatge. El moviment del marc del visor està sincronitzat amb el desplaçament de l'scroll efectuat a la imatge de treball. Si movem el marc del visor

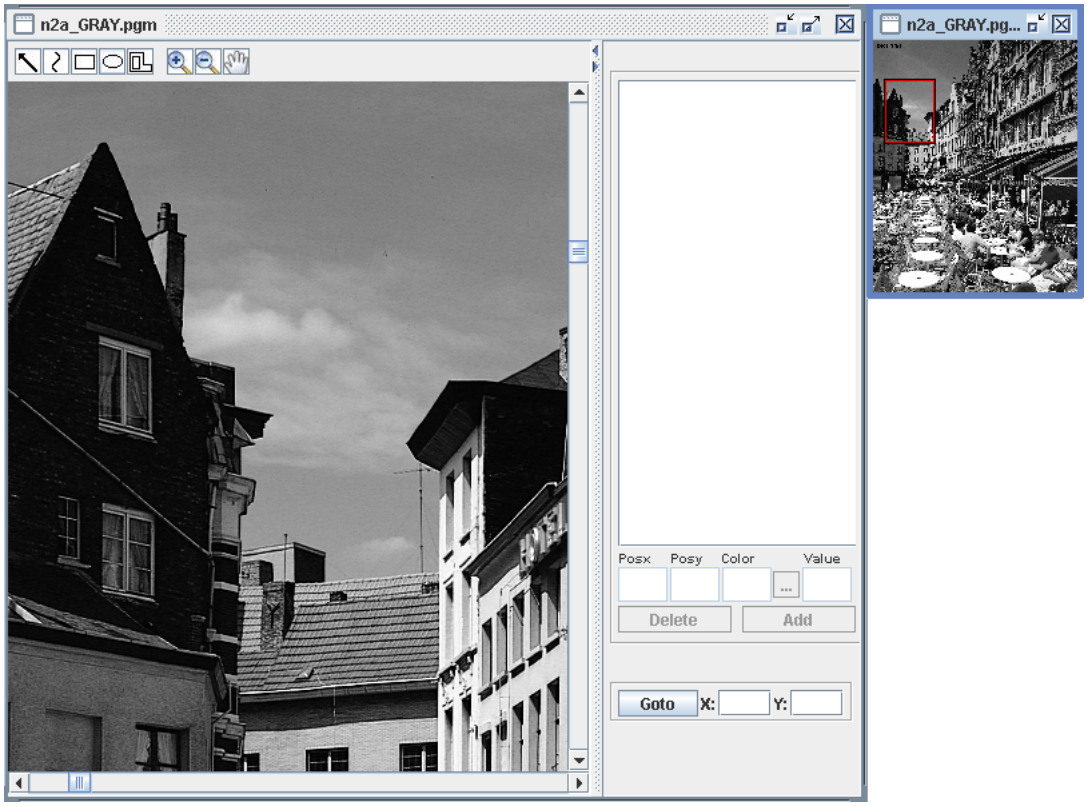


Figura A.10: Correspondència del visor amb la imatge.

(rectangle que podem veure a la figura A.10 en color vermell), veiem que es desplaça la imatge de la finestra de treball. I a l'inrevés, si movem la imatge de la finestra de treball, ja sigui amb el *panning* o amb l'scroll, veurem que se'ns resituarà el marc a la posició que li correspon. De la mateixa forma, si redimensionem la pantalla de treball, les dimensions del marc es redimensionaran també per adequar-se al nou tamany de la pantalla de treball. I si apliquem un nivell de zoom sobre la imatge de treball, d'igual manera és readaptarà el tamany del marc de visualització.

A.4 Com fer-ho

A aquesta secció explicarem pas per pas com fer les operacions més habituals amb l'aplicació per tal d'aclarir al màxim el seu funcionament.

A.4.1 Com dibuixar una regió

Per tal de dibuixar una regió hem d'executar l'aplicació i obrir la imatge amb la qual volem treballar. A partir d'aquí hem d'efectuar els següents passos:

1. Pitgem el botó *Draw*. Veiem que s'activa la selecció del color i el *textfield* del valor.

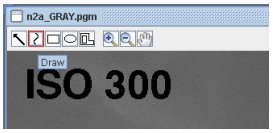


Figura A.11: Botó draw.

2. Seleccionem el color amb el que volem dibuixar la regió.

   (a) Premem el botó de selecció de color.

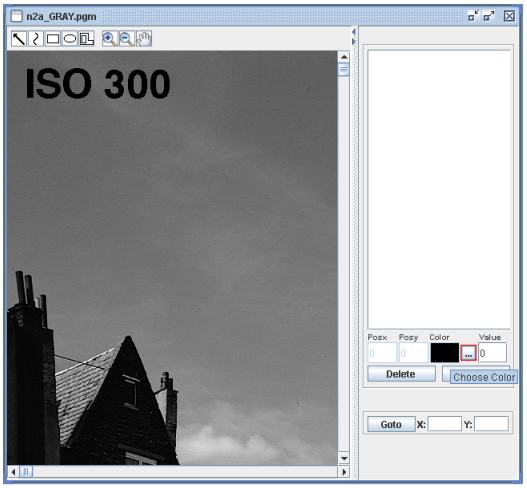


Figura A.12: Selecció del color.

   (b) Ens apareix la pantalla de selecció de color. Podem triar el color de la paleta, indicant el seu gradient o bé indicant directament el seu valor RGB.

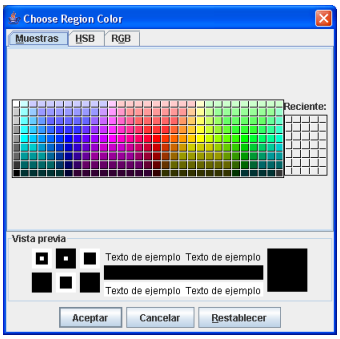


Figura A.13: Triem el color amb el qual volem dibuixar.

(c) Un cop triat el color, premem Acceptar. Veiem que apareix el color seleccionat al *textfield* de color.

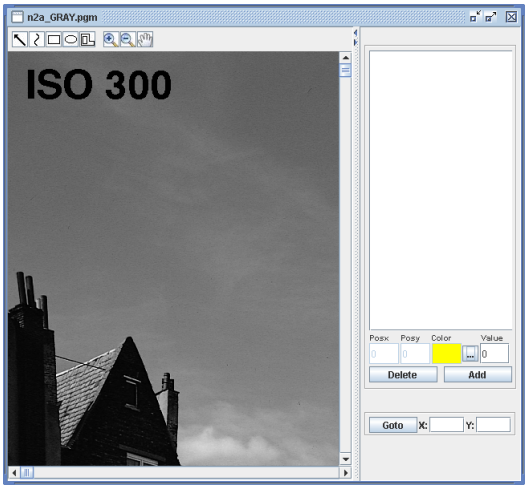


Figura A.14: Color seleccionat.

3. Comencem a dibuixar sobre la imatge prement el botó esquerra del *mouse*.

   Si la figura que volem dibuixar no es visualitza completament, no ens hem de preocupar. Quan ens apropem als marges de la figura, la figura es desplaçarà completament permetent-nos accedir a zones de la imatge que no ens eren visibles quan hem iniciat el dibuix de la figura. Cal tenir en compte que podem fer creuaments o interseccions a l'hora de dibuixar la figura. No és necessari que haguem de fer una figura "perfecte". A l'hora de calcular els punts que formen part de la regió, es tindrà en compte únicament els punts que queden dins d'una zona tancada.

4. La imatge ha de ser una imatge tancada, però aquest és un procés que realitza

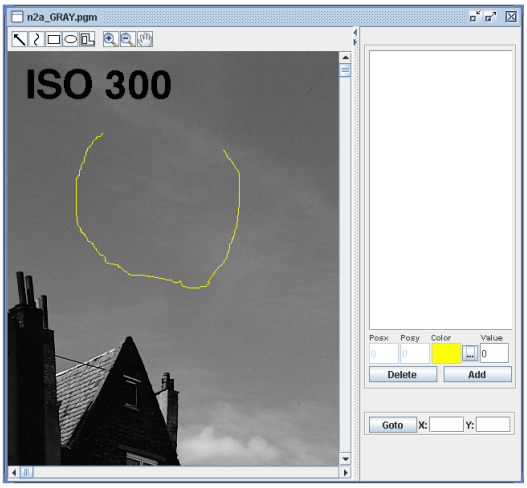


Figura A.15: Començant a dibuixar.

l'aplicació automàticament. Només ens hem d'apropar a una distància convenient del punt d'origen i al deixar anar el botó del *mouse*, la regió es tancarà automàticament.

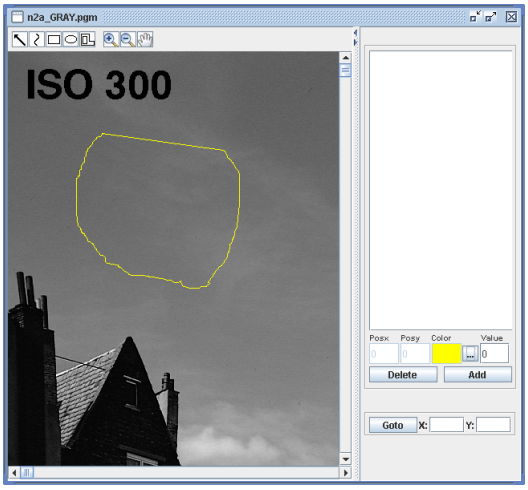


Figura A.16: Tancant la regió.

5. Indiquem el valor que volem per aquesta regió. El valor és el color amb el qual es guardarà la regió si desem la imatge com a màscara. El valor ha d'estar entre 1 i 255 (ambdós inclosos).

   Hem de diferenciar entre el color de la figura creada, que serveix per identificar visualment la figura que hem dibuixat, del valor amb que es guardarà la regió un cop l'haguem creada. El valor representarà el color de la regió un cop desem la imatge com a màscara.

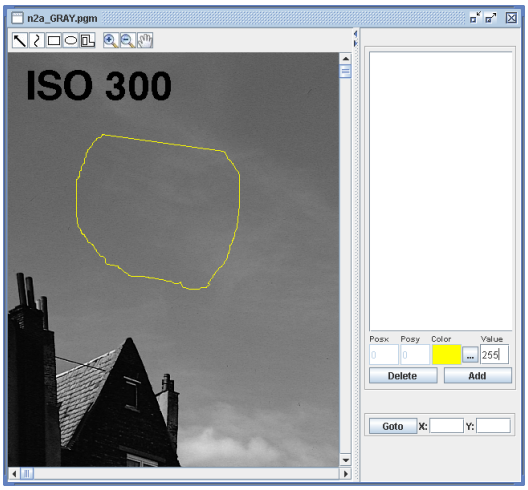


Figura A.17: Assignant el valor.

6. Premem el botó Add Region i s'afegeix la regió a la llista de regions. Veiem que es guarda la posició inicial (x0,y0), el color (R,G,B) i el valor triat.

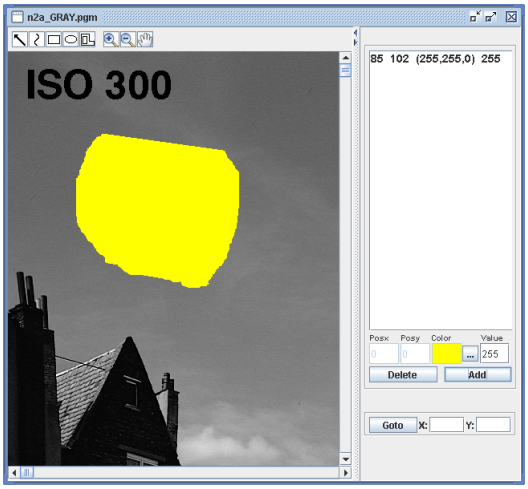


Figura A.18: Generant la regió.

La posició inicial (x0,y0) representa el punt en el qual hem iniciat el dibuix de la regió. S'ha afegit aquesta informació per ajudar a ubicar la posició de les regions a la imatge.

Si volem dibuixar una figura usant el botó *rectangle* o *el·lipse*, els passos a realitzar són exactament els mateixos. Només haurem de prémer el botó pertinent per iniciar un o altre dibuix.

A.4.2 Com crear una regió complexe

1. Pitgem el botó *Complex Shape*. Veiem que s'activa la selecció del color i el *textfield* del valor. Amb aquesta opció podrem dibuixar una figura composta formada per diferents regions.

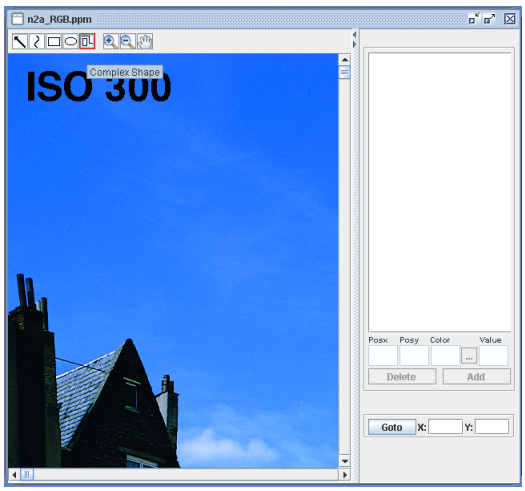


Figura A.19: Generant una figura composta.

2. Dibuixem una primera figura, seguint els mateixos passos que els indicats al punt 4.1. Cal tenir en compte, però, que si volem dibuixar una regió complexe, no podem prémer cap més botó de dibuix fins que finalitzem la regió que estem creant. En cas contrari perdríem el dibuix que estem fent.

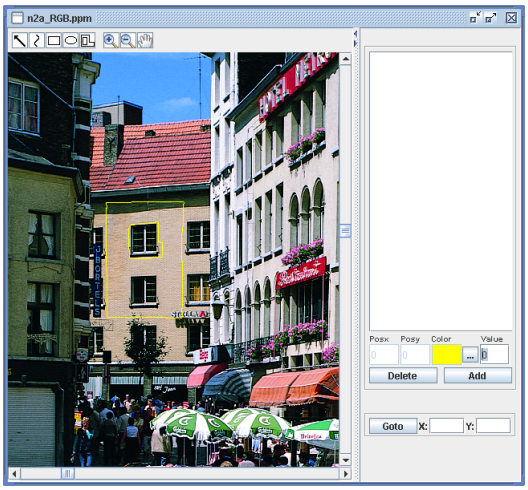


Figura A.20: Dibuixant una figura dins d'unaltre figura.

3. Un cop dibuixada una primera regió, podem tornar a dibuixar les figures que necessitem dins d'aquesta primera figura. Tantes com siguin necessàries.

4. Un cop dibuixada la figura composta que volem, assignem el valor i pitgem el botó *add region*.

Veiem a la figura A.21 que hem dibuixat una figura que agafa bona part de la

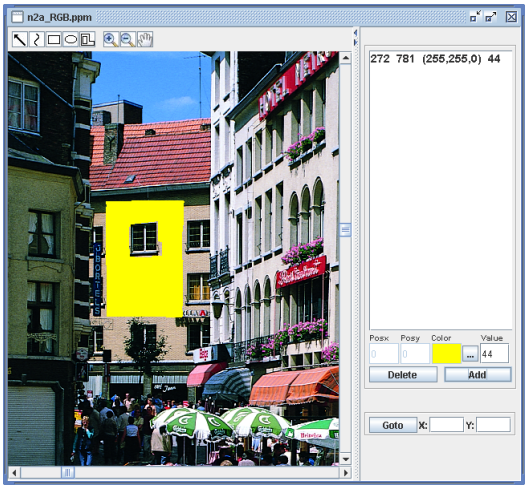


Figura A.21: Figura composta generada.

façana de la casa i dins d'aquesta figura hem dibuixat una altre figura seleccionant el contorn de la finestra. Quan hem generat la regió ha pintat tots els punts existents entre una i altre figura.

A.4.3 Com modificar una regió

Les regions generades ens apareixen a la llista situada a la dreta de la pantalla.

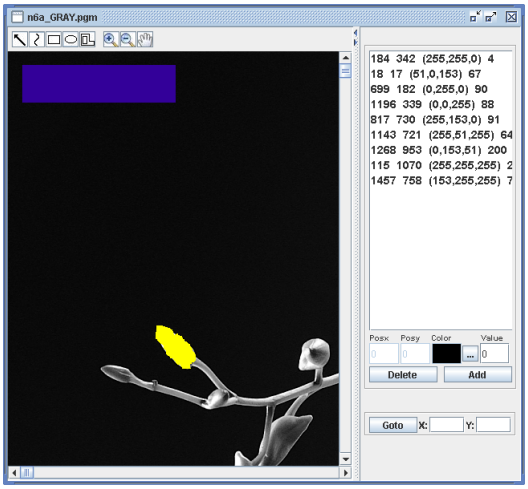


Figura A.22: Imatge amb diferents regions creades.

1. Quan hem generat diferents regions, les podem visualitzar totes a la llista que ens apareix a la dreta de la pantalla de treball. El que hem de fer és seleccionar amb el *mouse* una d'aquestes regions.

2. Automàticament apareixeran els valors d'aquesta regió als *textfields* de sota de la llista: posx, posy, color i valor. Els valors corresponents a posx i posy no els podem modificar, són únicament indicatius.

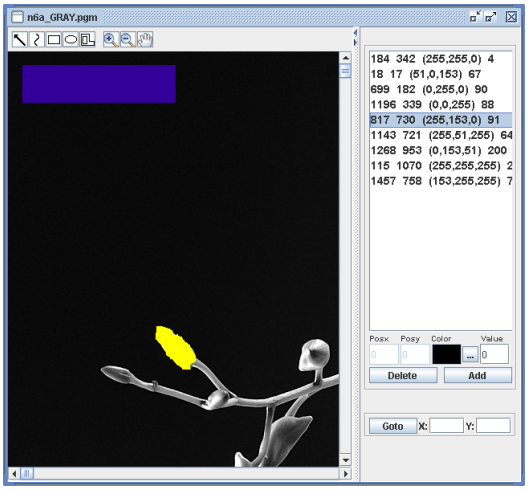


Figura A.23: Selecció d'una regió de la llista.

3. Podem modificar tant el valor com el color i els valors es refrescaran tant a la llista com al dibuix de la regió (en cas que haguem canviat el color).

A.4.4 Com esborrar una regió

Per tal de poder esborrar un regió, hem de tenir com a mínim una regió creada. En cas contrari el botó per esborrar regions es troba desactivat.

1. Seleccionem la regió que volem esborrar, pitjant-la amb el *mouse*.

2. Pitgem el botó *delete*. Veiem que la regió desapareix de la llista i que la figura associada desapareix de la imatge.

A.4.5 Com guardar una imatge

A l'hora de desar una imatge, tenim dues opcions. Generar una imatge PGM o bé PPM en funció de la imatge que tinguem carregada tal i com la veiem per pantalla o bé generar

una màscara de la imatge. Aquesta darrera opció és la que apareix configurada per defecte. La màscara consisteix en una imatge del mateix tamany que la imatge inicial però tota negre, només apareixeran les regions creades amb el color assignat al valor *value*. Tot seguit mostrem dues imatges generades a partir de la mateixa imatge font, la primera amb l'opció *Save Image with Regions* i la segona amb l'opció *Save Mask with Regions* activada.

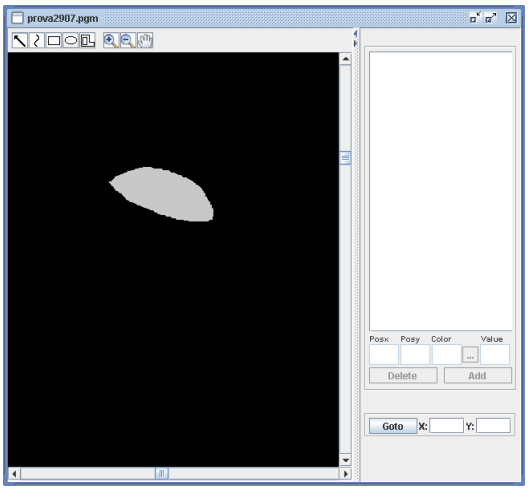


Figura A.24: Resultat de desar una imatge amb l'opció de màscara.

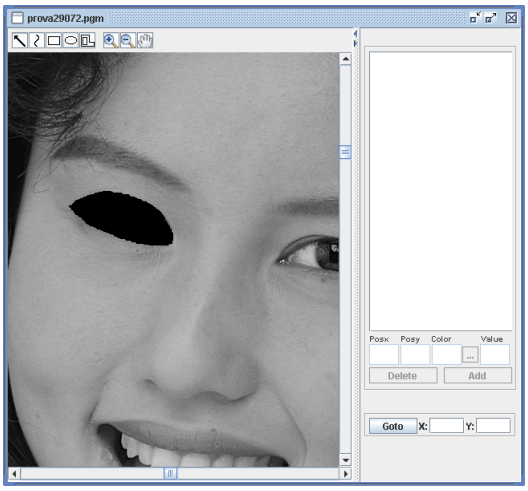


Figura A.25: Resultat de desar com a imatge.

Firmat: Joan Vidal i Plujà

Bellaterra, Setembre del 2007

Resum

En aquest projecte, titulat "GenRegion: aplicatiu de generació de màscares per a la codificació d'imatges amb regions d'interès", presentem una aplicació multiplataforma que visualitza imatges PGM i PPM amb un mínim cost de memòria i sobre les quals podrem dibuixar regions d'interès. L'aplicació carregarà només la part de la imatge que podem visualitzar a la pantalla de treball permetent treballar amb imatges de gran tamany. Les regions definides seran accessibles en tot moment, podent modificar-les i esborrar-les. Finalment podrem generar ja sigui una màscara de la imatge actual amb les regions definides o bé una imatge idèntica a l'original però on es visualitzaran les regions definides.

Resumen

En este proyecto, titulado "GenRegion: aplicación de generación de máscaras para la codificación de imagenes con regiones de interés", presentamos una aplicación multiplataforma que visualiza imágenes PGM y PPM con un mínimo coste de memoria y sobre las cuales podremos dibujar regiones de interés. La aplicación cargará únicamente la parte de la imágen que es posible visualizar en la pantalla de trabajo permitiendo trabajar con imagenes de gran tamaño. Las regiones definidas seran accesibles en todo momento, pudiendo modificarlas y borrarlas. Finalmente podremos generar ya sea una máscara de la imagen actual con las regiones definidas o bien una imagen idéntica a la original pero donde se visualizaran las regiones definidas.

Abstract

In this project, called "GenRegion: application of mask generation for the image with interesting regions codification", we presented a multiplatform application that visualizes PGM and PPM images with a minimum memory cost and where we will be able to draw interesting regions. The application will only load the region of the image that can be visualized in the work screen, allowing to work with images of great size. The defined regions would be accessible at any moment, being able to modify them and to erase them. Finally we will be able to generate a mask of the present image with the defined regions or an identical image to the original, but where the defined regions are showed.